A Library of High Level Control Operators *

Christian Queinnec†
École Polytechnique & INRIA-Rocquencourt

Abstract

Numerous high-level control operators, with various properties, exist in the literature. To understand or compare them is difficult since their definitions use quite different theoretical frameworks; moreover, to our knowledge, no implementation offers them all. This paper tries to explain control operators by the often simple stack manipulation they perform. We therefore present what we think these operators are, in an executable framework derived from abstract continuations. This library is published in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. For instance, we do not claim our implementation to be faithful nor we attempt to formally derive these implementations from their original definitions. The goal is to give a flavor of what control operators are, from an implementation point of view. Last but worth to say, all errors are mine.

Among the many existing control operators, we only consider `call/cc`, `prompt/control`, `shift/reset`, sequential `spawn`, `splitter/abort/call/pc`. We also add `dynamic-wind` to them although it does not belong to the same family of control operators.

Our *lingua franca* is Scheme extended with a simple class definition facility. We use Scheme *(i)* to avoid frightening Greek letters and, *(ii)* to provide implementations people can play with in any plain Scheme system. Our defining process is two-fold: first, we use a program transformation making continuations apparent: Abstract Continuation Passing Style (ACPS) [FWFD88]. ACPS is somewhat similar to Continuation Passing Style (CPS) but confers a richer structure to continuations. ACPS represents continuations by lists of "continuation slices". Implementors will recognize in these slices a functional abstraction for stack frames. In a second step, the rewritten equivalent program is evaluated with any regular Scheme evaluator. The key point is the use of ACPS which allows control operators to be written as regular (but reflective) Scheme functions acting on the representation of continuations. Our presentation has the definite advantage to be executable and to offer all these control operators altogether.

The paper has the following structure: we first present ACPS then exercise it on the well-known `call/cc` operator. This will be followed by a brief description of what partial continuations are. We will then expose the various sets of control operators[1].

1 Abstract Continuation Passing Style

CPS is a program transformation that makes continuations apparent. CPS also corresponds to a programming style where functions take an additional argument, the continuation: an unary function which is applied to the value that would have been returned in normal style. Continuations come from the work of Strachey and Wadsworth [SW74, Rey93] and were used to denote control features such as `goto` i.e., unconditional jump. CPS and sophisticated control handling have always been sources of inspiration, see [DL92, SF92, Mor92, FSDF93, Sit93] for recent developments.

Denotational techniques often drift into program transformations: CPS was exploited in Rabbit, the first Scheme compiler [Ste78], as a kind of intermediate language. CPS provides a simple (but not unique)

*`Id: contlib.bk,v 1.27 1994/04/11 08:10:43 queinnec Exp` . Submitted to Lisp Pointers.

†Laboratoire d'Informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France – Email: `queinnec@polytechnique.fr` This work has been partially funded by GDR-PRC de Programmation du CNRS.

1 Their definition is the actual code as it appears in the `contlib.tar.gz` package and automatically extracted using LiSP2TEX. See the Scheme Repository or `ftp.inria.fr:INRIA/Projects/icsla/Programs` to get them.

mean to compile programs using `call/cc`: CPS turns these programs into new equivalent programs where continuations are represented by regular closures which only require normal compiling technology[2]. In other terms, CPS turns `call/cc` into a regular function that the user could have directly written[3]:

```
(define (CPS-call/cc k1 f)
  (f k1 (lambda (k2 value) (k1 value))) )
```

In this definition, continuations appear as the first variable (this makes easier the introduction of dotted variables) with a name conventionally prefixed by `k`: functional applications therefore take the current continuation as first argument. Being a function, the reified continuation obeys the CPS protocol: it takes a continuation as first argument (`k2`) then forgets it and, finally applies `k1` to `value`. CPS-style continuations are closures i.e., opaque objects that can only be applied. More elaborate control operators need more information about the continuation.

Abstract continuations were introduced in [FWFD88] to describe the meaning of `prompt` and `control` operators. We use its syntactic counterpart and encode continuations as lists of frames. With an implementation bias, a frame will be represented by an object whose first slot contains a behavior i.e., a binary function that expects a stack (composed of all the frames that are below this very frame) and a value. When a frame is activated, it computes a new value to be sent back to the rest of the frames. Frames are represented by objects and not directly by behaviors so that they can be recognized and specialized to hold extra slots (other than behaviors), this will be needed later by some control operators.

Since the continuation is no longer represented by a function, we ought to provide the necessary procedures that encapsulate its representation: a value is sent to a continuation using `resume`. A frame is added, i.e., pushed, onto a continuation using `extend`.

```
(define-class frame Object (behavior))

(define (resume q v)
  ((frame-behavior (car q)) (cdr q) v) )
(define (extend q frame)
  (cons frame q) )
```

The `define-class` form defines a new class known as `frame` (subclass of the root class here named `Object`). A frame has a single slot that contains its `behavior`. This behavior can be extracted from a frame using the `frame-behavior` selector; `make-frame` builds a new frame out of a behavior; finally, a frame can be recognized with the `frame?` predicate. We use objects rather than records since we use inheritance in the `resume` function: the `frame-behavior` selector will thus work for any direct or indirect instances of `frame`[4].

As shown in the definition of `resume`, a behavior is a binary function taking a continuation and a value. A continuation is represented by a list of frames. To send a value to a continuation corresponds to the application of the behavior of the first frame to the rest of the frames and to the received value. This tightly mimics the behavior of a frame that receives a value and has access to the frames that are below it.

The ACPS program transformation appears in table 1, see [Que92b] for a variant. Computations are split into small atomic computation steps. A computation step is fed with a value and a continuation, may extend the continuation with new computations to be done or just gives a value back to the continuation. Making frames apparent allows some compile-time analyzes to compact the continuation or to remove some kinds of associatively wrapped recursions [Que93].

ACPS-transformed programs contain a lot of administrative redexes: elaborate algorithms [DF92, SF92] can be devised to avoid their generation. One can also observe that CPS may be recovered from ACPS with a special encoding of `resume` and `extend`:

```
(define (resume q v) (q v))
(define (extend q frame)
  (lambda (v) ((frame-behavior frame) q v)) )
```

To project an ACPS-transformed program into a CPS-transformed program i.e., to transform inspectable lists of frames into opaque closures, restrict the possibility of introspection which is the basis of reflective programming [dRS, FW84, Wan86, dR87, WF88, DM88].

[2] To compile *well* these closures requires a higher compiling technology [App92].
[3] This is true only if the user can submit such a definition and prevent the evaluator to CPS-convert it.
[4] We use our own home-made object system called MeroonV3, consult again the Scheme repository to get it.

$\mathcal{ACPS}[\![\nu]\!]q =$ (resume q ν) *if ν is a non global variable*
$\mathcal{ACPS}[\![\nu]\!]q =$ (resume q ACPS-ν) *if ν is a global variable*
$\mathcal{ACPS}[\![$(quote ε)$]\!]q =$ (resume q (quote ε))
$\mathcal{ACPS}[\![$(if π_0 π_1 π_2)$]\!]q = \mathcal{ACPS}[\![\pi_0]\!]$(extend q (make-frame (λ(q' v') (if v' $\mathcal{ACPS}[\![\pi_1]\!]q'$ $\mathcal{ACPS}[\![\pi_2]\!]q'$))))
$\mathcal{ACPS}[\![$(set! ν π)$]\!]q = \mathcal{ACPS}[\![\pi]\!]$(extend q (make-frame (λ(q' v') (resume q' (set! ν v')))))
$\mathcal{ACPS}[\![$(λ(v^*) π)$]\!]q =$ (resume q (λ(q' v^*) $\mathcal{ACPS}[\![\pi]\!]q'$))
$\mathcal{ACPS}[\![$(π_1 ... π_n)$]\!]q_0 =$
  $\mathcal{ACPS}[\![\pi_1]\!]$(extend q_0 (make-frame (λ(q_1 v_1) ...
    $\mathcal{ACPS}[\![\pi_n]\!]$(extend q_{n-1} (make-frame (λ(q_n v_n) (v_1 q_n v_2 ... v_n))))...)))

Table 1: ACPS rules

To end this section, we give the usual factorial function as transformed by ACPS and massaged a little to avoid administrative redexes.

```
(DEFINE FACT (LAMBDA (N) (IF (= N 1) 1 (* N (FACT (- N 1))))))
→ (DEFINE ACPS-FACT
     (LAMBDA (Q73 N)
       (IF (= N 1)
           (RESUME Q73 1)
           (ACPS-FACT (EXTEND Q73 (MAKE-FRAME
                                   (LAMBDA (Q81 V78)
                                     (RESUME Q81 (* N V78)) ) ))
                      (- N 1) ) ) ) )
```

An interesting point of this massaged form is related to errors like trying to multiply a non-number. If the error handling system allows you to correct this on the fly, then obviously the multiplication has a mean to reify and use (infra-)continuations that do not appear in the above program.

The continuation of the internal recursive call to `ACPS-fact` explicitly shows the pending multiplication frame, [Que93] takes benefit of this static observation to combine frames together thus opening new opportunities for optimizations. ACPS is very expressive and can be used to describe implementations of dynamic variables, error handling, multitasking [Que92b, Que92a, QD93].

2 *À tout seigneur, tout honneur:* `call/cc`

The `call/cc` operator is the basic control operator of Scheme. It reifies the current continuation into a function and applies its sole argument on it, the value of this application becomes the value of the original `call/cc` form. `call/cc` can be used to program any sequential control operator and, at least, multitasking [Wan80], engines [HF84], escapes [HFW84] and even partial continuations! From a more implementational point of view, three usages can be recognized:

- escape operator: the continuation is only invoked once and only while in the dynamic extent of the `call/cc` form that created it. This usage provides a `setjmp/longjmp` facility à la C, `catch/throw` and variants à la Common Lisp, `call/ep` à la PaiLisp [IM89] etc.
- coroutine operator: the computation is reified into a continuation, which will be invoked at most once later. An example of coroutine scheduler appear in [Wan80, Mat92];
- the last case is when a continuation is multiply invoked i.e., multiple values are returned to the same continuation. This can be used by generators à la Icon [MH90] but also appears in presence of concurrency [QD93].

The definition of `call/cc` as defined in [CR91] and according to ACPS, is as simple as it was according to CPS. Observe the duplication of `q1` which is still the current continuation of the call to `f` as well as a part of the reified continuation. This apparent simplicity must not hide the fact that continuations are difficult to compile and implement efficiently as can be seen in [CHO88, HD90, JG89, Mat92].

```
(define (ACPS-call/cc q1 f)
  (f q1 (lambda (q2 value) (resume q1 value))) )
```

Figure 1: `call/cc`

3 Partial continuations

Consider the following function, which uses `call/cc` to perform a premature exit. It receives a list of numbers to multiply:

```
(define (multiply-list list-of-numbers)
  (call/cc (lambda (exit)
             (define (mult l)
               (if (null? l) 1
                   (if (= (car l) 0) (exit 0) (* (car l) (mult (cdr l)))) ) )
             (mult list-of-numbers) )) )
```

Whenever a zero is found, an immediate escape is performed to directly return zero without performing any multiplications. Suppose we invoke `multiply-list` on `(4 3 2 0 ...)`, to escape means that, when zero is encountered, a prefix of the continuation (the prefix that is waiting for a value to multiply by `2`, `3` and `4`) will suddenly disappear. Rather than just erasing this prefix from the stack, one might imagine to turn it into an interesting programmatic object: this is the essence of partial continuations. Still using the previous example, the partial continuation represents three pending multiplications and is equivalent to $\lambda x.4*(3*(2*x))$.

Once reified, a partial continuation can be used as a regular function. Unlike continuations, partial continuations return a value and therefore can be composed. A partial continuation is a "continuation slice" which may be viewed as the "difference" of two regular continuations [MQ93]. A partial continuation is thus identified by two points in a stack and all partial continuation control operators offer the three following functionalities:

1. to identify where a partial continuation starts,
2. to identify where a partial continuation finishes,
3. to reify a partial continuation between these two points.

Very often, the two last actions are not separated, they are performed in one go.

Partial continuations appeared in [FFDM87, FFDM87, FF87] as well as in [Joh87, JD88]. Many papers follow in the Scheme realm and among them [Fel88, DF90, HD90, QS91, MQ93]. A detailed example of partial continuation use appears in [Dan89].

4 Useful functions

We introduce some functions to ease the forthcoming definitions. `extend*` takes a list of frames whereas `extend` takes only one frame. The `split` operator takes a continuation `q`, a predicate, a success function and a failure thunk. It looks for the first tail of `q` that satisfies the predicate. If such a tail is found then the success function is called on this tail as well as on the list of frames that precede the found tail; otherwise, the failure thunk is invoked.

```
(define (extend* q frames)
  (if (pair? frames)
      (extend (extend* q (cdr frames)) (car frames))
      q ) )
(define (split q predicate success failure)
  (define (scan q frames)
    (if (pair? q)
        (if (predicate q)
            (success (reverse frames) q)
            (scan (cdr q) (cons (car q) frames)) )
        (failure) ) )
  (scan q '()) )
```

The control operators are generally programmed as follow: the beginning of a partial continuation is identified by a special frame pushed onto the stack. When a partial continuation needs to be reified, the `split` function is used on the current continuation to find the frames that are above this special frame. These frames, properly wrapped with a functional interface, represent the partial continuation.

The definitions below do not take efficiency into account but they try to suggest that there exist common patterns.

5 `prompt, control`

Two control operators, `prompt` and `control`, were presented and discussed in [Fel88, FWFD88, SF90a, SF90b]. The `prompt` operator identifies contexts to be used by `control`. The `control` function reifies the context up to the nearest dynamically enclosing `prompt`, into a partial continuation and, at the same time, removes it from the current continuation. `prompt` is a syntax which expands into a call to `prompt-evaluate`; remember that `prompt-evaluate` is ACPS-transformed into `ACPS-prompt-evaluate`.

To define these control operators, we use a subclass of `frame`: `prompt-frame`, to mark the presence of a prompt; its behavior is similar to the identity, as represented by `resume`: it just passes the value it receives back to the stack below it. `prompt-evaluate` pushes a prompt-frame onto the stack and then invokes its first argument: a thunk. When invoked, `control` unwinds the frames of the current continuation until finding a prompt-frame, wraps all unwound frames into a regular function and applies its argument on it.

```
(define-class prompt-frame frame ())
(define-syntax prompt
  (syntax-rules ()
    ((_ expression) (prompt-evaluate (lambda () expression))) ) )

(define (ACPS-prompt-evaluate q thunk)
  (thunk (extend q (make-prompt-frame resume))) )
(define (ACPS-control q f)
  (split q (lambda (q) (prompt-frame? (car q)))
         (lambda (frames q)
           (f q (lambda (q v)
                  (resume (extend* q frames) v) )) )
         (lambda () (error "No enclosing prompt")) ) )
```

Figure 2: `prompt/control`

5.1 Examples

Here are some simple examples of `prompt/control` that illustrate various points. More comprehensive examples are given in the above references.

```
(prompt (* 2 (control (lambda (f) 3))))             → 3      [a]
(prompt (* 2 (control (lambda (f) (* 5 (f 3))))))   → 30     [b]
(prompt (* 2 (control (lambda (f) (f (f 3))))))     → 12     [c]
((prompt (* 2 (control (lambda (f) f)))) 3)         → 6      [d]
```

In these four examples, the partial continuation is $\lambda x.2{*}x$. When `control` reifies this partial continuation, it removes its associated frames from the stack as shown in example *[a]*. This is an "abort", an imperative effect. The partial continuation is a regular function that yields a value, as shown in *[b]*; hence partial continuations are composable, see example *[c]*. Once reified, there is no restriction on the use of partial continuations: they can be used out of the extent of the control form that creates them as illustrated on example *[d]*.

These examples are simple and show the interest of partial continuations. Unfortunately, there are more complex usages when control operators are called from within the reified continuation. It is difficult to figure out what is natural in these cases and we will see that it is a divergence point between the various sets of control operators.

```
(prompt (* 5                                                          [e]
           (prompt (* 2
                      (control (lambda (f2)              ;f2= λx.2 * x
                                 (* 3 (control (lambda (f3) ;f3= λx.3 * x
                                                  7 ))) )) )) ))  → 35
(prompt (* 5 (                                                        [f]
              (prompt (* 2
                         (control (lambda (f2)                ;f2= λx.2 * x
                                    (lambda ()
                                      (* 3 (control (lambda (f3) ;f3= λx.5 * (3 * x)
                                                       7 ))) ) )) )) )))
                                                  → 7

(prompt (* 5                                                          [g]
           ((lambda (x)                 ;let φ be this abstraction
              (control (lambda (f1)     ;f1= λv.2 * 5 * v
                         x )) )
            (* 3 (control (lambda (f2)  ;f2= λu.5 * φ(3 * u)
                            (* 2 (f2 7)) ))) ) ))  → 21
```

Example *[e]* exhibits the effect of a `control` inside a `control`, the two `control` forms reify a partial continuation up to the same `prompt`. On the contrary, example *[f]* shows embedded `control` forms reiying up to different `prompt` forms. The final example *[g]* shows `control` called from inside the application of a reified partial continuation. These examples are complex and require some care to be checked.

5.2 `call/cc` with `prompt/control`

The `control` operator is somewhat dynamic since it looks for the nearest enclosing prompt-frame in the continuation. This dynamic behavior of `control` poses problems of capture similar to those posed by dynamic variables in traditional Lisp interpreters. To arbitrarily wrap an expression into a `prompt` form i.e., to replace π by `(prompt` π`)` may thus have a non-local effect. Using multiple embedded `control` forms is difficult but can be done if some protocol is added around each `prompt` form as studied in [SF90a].

The `call/cc` operator can be defined with these operators. Given a program π not using `prompt`, we can provide the `call/cc` facility, rewriting π as:

```
(let ()
  (define (call/cc f)
```

```
    (control (lambda (pc)
               (pc (f (lambda (v)
                        (control (lambda (ignore-pc) (pc v))) ))) )) )
  (prompt π) )
```

Reciprocally, `prompt` and `control` can be simulated with `call/cc` (and assignment) as explained in [SF90a].

6 shift/reset

Danvy and Filinski introduced in [DF90, DF89] a new couple of control operators: `reset` and `shift`; `reset` identifies the contexts up to which `shift` reifies partial continuations. A `shift` form reifies its context up to the dynamically nearest `reset` form and removes it from the current continuation. `reset` and `shift` are syntaxes that we expand into functions taking thunks as argument: `reset-evaluate` and `shift-evaluate`.

One interesting difference with `prompt/control` lies in the definition of the reified partial continuation: it now contains (and finishes with) the nearest reset-frame. Therefore partial continuations have at least their invocation point as enclosing context.

```
(define-class reset-frame frame ())
(define-syntax reset
  (syntax-rules ()
    ((_ expression) (reset-evaluate (lambda () expression))) ) )
(define-syntax shift
  (syntax-rules ()
    ((_ variable expression)
     (shift-evaluate (lambda (variable) expression)) ) ) )

(define (ACPS-reset-evaluate q thunk)
  (thunk (extend q (make-reset-frame resume))) )
(define (ACPS-shift-evaluate q f)
  (split q (lambda (q) (reset-frame? (car q)))
         (lambda (frames q)
           (let ((frames (append frames (list (car q)))))
             (f q (lambda (q v)
                    (resume (extend* q frames) v) )) ) )
         (lambda () (error "No enclosing reset")) ) )
```

Figure 3: `reset/shift`

6.1 Examples

Here are the previous examples rewritten with these new control operators. Examples were similarly named to ease comparison. There are no divergence in the six first examples, from *[a]* to *[f]*.

```
(reset (* 2 (shift f 3)))            → 3                         [a]
(reset (* 2 (shift f (* 5 (f 3))))) → 30                         [b]
(reset (* 2 (shift f (f (f 3)))))   → 12                         [c]
((reset (* 2 (shift f f))) 3)       → 6                          [d]
(reset (* 5                                                      [e]
          (reset (* 2 (shift f2              ;f2= λx.reset(2*x)
                             (* 3 (shift f3 ;f3= λx.reset(3*x)
                                        7 ))) )))) ))  → 35
(reset  (* 5 (                                                   [f]
```

```
                    (reset (* 2
                             (shift f2                ;f2= λx.reset(2 * x)
                                   (lambda ()
                                     (* 3 (shift f3 ;f3= λx.reset(5 * 3 * x)
                                                 7 )) )) )) )))  → 7
      (reset (* 5                                                                [g]
               ((lambda (x)          ;let φ be this abstraction
                  (shift f1 x) )     ;f1= λv.reset(5 * v)
                (* 3 (shift f2       ;f2= λu.reset(5 * φ(3 * u))
                       (* 2 (f2 7)) ))  )  ))  → 42
```

A difference can be observed in the seventh example *[g]*. The reason is that when the partial continuation `f2` is invoked, the `(shift f1 x)` form reifies up to the nearest reset-frame and thus do not capture the pending multiplication by 2 since the call to `f2` reinstalls a reset-frame.

6.2 `call/cc` with `shift/reset`

As for the other control operators, it is possible to program `call/cc` with `shift` and `reset`. Once again, the program π is required not to use `reset` to avoid interference with the coding of `call/cc`. The program π is wrapped as:

```
(let ()
  (define (call/cc f)
    (shift pc (pc (f (lambda (v)
                       (shift ignore-pc (pc v)) )))) )
  (reset π) )
```

7 `spawn`

Hieb and Dybvig presented, in [HD90], a new control operator called `spawn`. Although specified in presence of concurrency, `spawn` is useful even in the absence of multiple processes. We will only present a sequential version here. One of the goals of the authors was to replace the dynamic behavior of the two previous proposals by more control on the point up to which a partial continuation is reified. A consequence is that `spawn` reifies a *process controller* that can only be used while in the dynamic extent of the `spawn` form that created it. The reason is that if you want to reify up to a `spawn` point, this point must exist somewhere in your future i.e., you must be in the dynamic extent of this very `spawn` form. This problem does not exist with `control` and `shift` since they do not target a specific point but just the nearest one, whatever it is. Finally, the authors devised a functional protocol with a single special function, `spawn`, to offer the identification of the context up to which reification is needed and the identification of the reification point.

`spawn` takes a single argument, a unary function that will be invoked by `spawn` on a synthesized object named a process controller, say c. The process controller c can only be applied while in the dynamic extent of the original `spawn` form. Whenever applied, it takes a single argument: an unary function that it applies on the reified partial continuation that extends up to the associated `spawn`; at the same time the process controller removes this partial continuation from the current continuation. Similarly to `shift`, the reified partial continuation retains the spawn-frame up to which it is reified. `spawn` can be viewed as a generalized `prompt` creating a new `control` function at each invocation.

7.1 Examples

To give a flavor of `spawn`, let us once again rewrite the former examples. With respect to the previous section and roughly said, `reset` is changed into `(spawn (lambda (`c`) ...))` while `(shift f ...)` is replaced by `(`c `(lambda (f) ...))`. This works well if there is a single `reset` form but if there are more, then we can choose up to which one reification is wanted, see for instance, examples *[f1]* and *[f2]*.

```
(spawn (lambda (c) (* 2 (c (lambda (f) 3)))))          → 3              [a]
```

```
(define-class spawn-frame frame ())

(define (ACPS-spawn q f)
  (let ((frame (make-spawn-frame resume)))
    (f (extend q frame)
       (lambda (q g)
         (split q (lambda (q) (eq? (car q) frame))
                (lambda (frames q)
                  (let ((frames (append frames (list (car q)))))
                    (g q (lambda (q v)
                           (resume (extend* q frames) v) )) ) )
                (lambda () (error "Out of extent")) ) ) ) ) )
```

Figure 4: spawn

```
(spawn (lambda (c) (* 2 (c (lambda (f) (* 5 (f 3)))))))  → 30      [b]
(spawn (lambda (c) (* 2 (c (lambda (f) (f (f 3)))))))    → 12      [c]
((spawn (lambda (c) (* 2 (c (lambda (f) f)))) 3))        → 6       [d]
(spawn (lambda (c1)                                                [e]
         (* 5 (spawn (lambda (c2)
                  (* 2 (c2 (lambda (f2)            ;f2= λx.2*x
                     (* 3 (c2 (lambda (f3) ;f3= λx.3*x
                                7 ))) ))) ))) ))  → 35
(spawn (lambda (c1)                                                [f1]
         (* 5 ((spawn (lambda (c2)
                  (* 2 (c2 (lambda (f2)
                          (lambda ()
                            (* 3 (c1 (lambda (f3)
                                       7 ))) ) ))) )))) ))  → 7
(spawn (lambda (c1)                                                [f2]
         (* 5 ((spawn (lambda (c2)
                  (* 2 (c2 (lambda (f2)
                          (lambda ()
                            (* 3 (c2 (lambda (f3)
                                       7 ))) ) ))) )))) ))
```
→ *Out of extent error,* c2 *is not used in its correct dynamic extent*

```
(spawn (lambda (c)                                                 [g]
         (* 5 ((lambda (x) (c (lambda (f1) x)))
               (* 3 (c (lambda (f2) (* 2 (f2 7))))) )) ))     → 42
((spawn (lambda (c1)                                               [h]
          (* 5 (spawn (lambda (c2)
                  (* 2 ((c1 (lambda (f1) ;f1=λu.(5*2*(u))(φ)
                           (lambda (g) (g f1 c2)) )))) ))) ))
 ;;let φ be the value of the following term. It will be bound to g.
 (lambda (f1 c2) (f1 (lambda () (* 7 (c2 (lambda (f2) ;f2=λv.2*7*v
                                   (* 11 (f2 13)) )))))) )
       → 10010   i.e., 5*11*2*7*13
```

Example *[g]* still yields the same value; although there are two spawn-frames associated to c in the stack, spawn chooses to reify a partial continuation up to the closest. Therefore it does not capture the pending multiplication by two.

The final example is the most mind-boggling, it shows a fine example where the concept of dynamic extent appears not so intuitive. In example *[h]*, a partial continuation `f1` and a process controller `c2` are used far from their birth site. `f1` reinstalls the spawn-frame associated to `c2` that it captured when created. Consequently, to call `c2` within `f1`, is correct since we are still in its dynamic extent. We will return to the concept of dynamic extent in section 8.2.

7.2 `call/cc` with `spawn`

Similarly to the previous operators it is possible to write `call/cc` with `spawn`. This is even easier since now there is no restriction on programs which can use `spawn` for themselves without interference. The reason is that now `call/cc` captures the correct process controller:

```
(spawn (lambda (lower)
         (define (call/cc f)
           (lower (lambda (pc)
                    (pc (f (lambda (v)
                             (lower (lambda (ignore) (pc v))) ))) )) )
         π ))
```

8 `splitter/abort/call/pc`

Another set of control operators was proposed by Queinnec and Serpette in [QS91][5]. Their effect is very close to `spawn` although one goal was to separate the different effects involved by `spawn` into different operators. `splitter` marks a context up to which partial continuations can be reified. The context is reified into an object called a mark which can serve as first argument to the `abort`, `call/pc` and `within-extent?` functions. `abort` allows to replace the current computation up to a given mark with a new computation expressed with a thunk. `call/pc` reifies the partial continuation up to the given mark and applies its second argument on it. In contrast to the previous `control`, `shift` or process controller, `call/pc` only reifies the partial continuation but does not remove it: this is the job of `abort` which is then the only imperative control operator. Similarly to `spawn`, marks can only be used while in the dynamic extent of the `splitter` form that created them and this can be tested with the `within-extent?` predicate adding some reflective capability to the language. Finally and similarly to `control`, a reified continuation does not capture the splitter-frame up to which it was reified.

8.1 Examples

Since `splitter` provides more basic operators, our preferred examples become more verbose. Compared to the previous section and again roughly stated, `(spawn (lambda (`*c*`) ...))` is rewritten as `(splitter (lambda (`*c*`) ...))` while `(`*c* `(lambda (f) ...))` is replaced by `(call/pc `*c* `(lambda (f) (abort `*c* `(lambda () ...))))` to combine the reification and imperative abortion effect of process controller.

```
(splitter (lambda (m) (* 2 (call/pc m (lambda (f)                            [c]
                                        (abort m (lambda ()
                                                   (f (f 3)) )) ))))) → 12
(splitter (lambda (m) (* 2 (call/pc m (lambda (f) (f (f 3))))))) → 24        [c1]
((splitter (lambda (m) (* 2 (call/pc m (lambda (f)                           [d]
                                         (abort m (lambda () f)) )))))
 3 )   → 6
(splitter                                                                    [e1]
 (lambda (m1)
   (* 5 (splitter
         (lambda (m2)
           (* 2 (call/pc m2 (lambda (f2)                    ;f2= λu.2 * u
```

[5] The presentation of [QS91] is slightly different, we adopt here one of the described variants.

```
(define-class splitter-frame frame ())
(define-class mark Object (q))

(define (ACPS-splitter q f)
  (let* ((frame (make-splitter-frame resume))
         (q (extend q frame)) )
    (f q (make-mark q)) ) )
(define (ACPS-within-extent? q mark)
  (resume q (split q (lambda (q) (eq? q (mark-q mark)))
                   (lambda (frames q) #t)
                   (lambda () #f) )) )
(define (ACPS-abort q mark thunk)
  (split q (lambda (q) (eq? q (mark-q mark)))
         (lambda (frames q) (thunk q))
         (lambda () (error "Out of extent")) ) )
(define (ACPS-call/pc qq mark f)
  (split qq (lambda (q) (eq? q (mark-q mark)))
         (lambda (frames q)
           (f qq (lambda (q v)
                   (resume (extend* q frames) v) )) )
         (lambda () (error "Out of extent")) ) )
```

Figure 5: `splitter`

```
                           (* 3 (call/pc m2 (lambda (f3) ;f3= λu.(2*(3*u))
                                                 7 ))) ))) ))) ))
                    → 210   i.e., 5*2*3*7
(splitter (lambda (m)                                                          [g]
            (* 5 ((lambda (x) (call/pc m (lambda (f1) (abort m (lambda () x)))))
                  (* 3 (call/pc m (lambda (f2)
                                    (abort m (lambda () (* 2 (f2 7)))) ))) )) ))
                    → 21
((splitter (lambda (m1)                                                        [h]
             (* 5 (splitter
                   (lambda (m2)
                     (* 2 ((call/pc m1 (lambda (f1) ;f1=λu.(5*2*(u))(φ)
                                         (abort m1 (lambda ()
                                                     (lambda (g) (g f1 m2))
                                                     )) )))) ))) ))
 ;;let φ be the value of the following term. It will be bound to g.
 (lambda (f1 m2) (f1 (lambda ()
                       (* 7 (call/pc m2 (lambda (f2) (* 11 (f2 13))))) ))) )
        →   Error: out of extent wrt m2
```

The major differences with `spawn` are *(i)* `splitter` always refers to the lowest associated splitter-frame whereas `spawn` prefers the nearest one, see example *[g]*. Partial continuations can in effect duplicate frames which can therefore appear more than once and `splitter` always refer to the lowest i.e., the one which has been associated to the mark once created. *(ii)* When applied out of the extent of the corresponding `splitter` form a partial continuation cannot call again `call/pc` nor `abort` as shown in example *[h]* above.

8.2 Dynamic extent revisited

Dynamic extent is often associated to the lifetime of a computation: dynamic extent ends when the computation returns a value. In presence of high level control operators such as `call/cc` (and mainly the third effect as mentioned in section 2), determining when a computation ends is difficult (GC can help). At least two views of dynamic extent suggest themselves.

- *dynamic extent wrt to a continuation:* A computation is within the dynamic extent of a continuation `q` if the current continuation has `q` as tail, this is what checks `within/de?` with `eq?`.
- *dynamic extent wrt to a frame:* A computation is within the dynamic extent of a frame if that frame belongs to the current continuation. This is the method `spawn` uses.

The first definition for dynamic extent has the property that when exited, it can be no more entered again. The second allows the dynamic extent to contain "holes", portions of time where the dynamic extent is left.

Example *[h]* exhibits that difference. With `spawn`, dynamic extent wrt `c2`, is captured by `f1` and is reinstalled when `f1` is called so `c2` can be used to reify `f2`. With `splitter`, the dynamic extent of `m2` is finished as soon as `(abort m1 ...)` is performed, so trying to reify up to `m2` is no longer possible.

Interestingly, dynamic extent requires the implementation of `spawn` and `splitter` to use `eq?` on frames or continuations i.e., needs a store.

8.3 `call/cc` with `splitter/call/pc/abort`

Following the tradition, writing `call/cc` with `splitter` is possible. This implementation explicitly shows the copy of the interesting stack-slice `pc` inside the reified continuation. Observe also that `call/pc` does not remove the reified partial continuation so it is not necessary to call it immediately to reestablish the correct current continuation. Interestingly, the imperative effect performed when invoking continuations is explicit due to the presence of `abort` in the reified continuation. A definition of `splitter` in terms of `call/cc` appears in [QS91].

```
(splitter (lambda (lower)
            (define (call/cc f)
              (call/pc lower (lambda (pc)
                               (f (lambda (v)
                                    (abort lower (lambda () (pc v))) )) )) )
            π ))
```

9 `dynamic-wind` and `call/cc` *nouveau*

The Scheme community has a sort of an equivalent of COMMON LISP's `unwind-protect` often named `dynamic-wind`. This form takes three thunks: a prelude, something to do (the real work) and a postlude. The idea is to execute the postlude whenever control escapes from the real work and to execute the prelude whenever control enters again the real work. An operational description can be found in [HF87, FWH92], ACPS allows us to give another one, more functional (without side-effect) and more compact:
Many implementation choices exist in this definition: one can substitute, for instance, `(work q2)` by `(work q1)` or, `(resume q3 value)` by `(resume q value)` whether one wants to make implementations more efficient or real work less sensitive to the side effects on control that can be performed by pre- or post-ludes. For instance, to write `(resume q value)` implies that the continuation is always the original caller of the `dynamic-wind` form whereas `(resume q3 value)` implies that the continuation of the postlude will be the frames beneath. In general, this is the original caller of the `dynamic-wind` form unless the frame is copied via partial continuation reification and reinstalled elsewhere.

The `dynamic-wind` operator only pushes a wind-frame, therefore a new `call/cc` has to take these frames into account. This new `call/cc` now creates continuations that, when invoked, unwind the current continuation to execute all postludes until finding a point where the target continuation can be rewound executing all necessary preludes.

```
(define-class wind-frame frame (prelude postlude))

(define (ACPS-dynamic-wind q prelude work postlude)
  (let ((q1 (extend q (make-wind-frame
                       (lambda (q4 value)
                         (postlude
                          (extend q4
                                  (make-frame
                                   (lambda (q3 ignore-postlude-value)
                                     (resume q3 value) ) ) ) ) ) )
                       prelude
                       postlude ) )))
    (prelude (extend q1 (make-frame
                         (lambda (q2 ignore-prelude-value)
                           (work q2) ) ))) ) )
```

Figure 6: `dynamic-wind`

```
(define (ACPS-new-call/cc qq function)
  (define (rewind q value)
    (define (deepest-wind qq) ;find the deepest wind-frame in qq till q
      (if (eq? qq q)
          #f
          (or (deepest-wind (cdr qq))
              (if (wind-frame? (car qq)) qq #f) ) ) )
    (let ((wq (deepest-wind qq))) ;evaluate preludes from q to qq
      (if wq ((wind-frame-prelude (car wq))
              (extend (cdr wq)
                      (make-frame
                       (lambda (ignore-q ignore-prelude-value)
                         (rewind wq value) ) ) ))
          ;;resume the caller of new-call/cc
          (resume qq value) ) ) )
  (define (unwind q value)
    (define (tail? q1 q2) ;is q1 a tail of q2 ?
      (or (eq? q1 q2)
          (and (pair? q2) (tail? q1 (cdr q2))) ) )
    (if (tail? q qq) ;is q an ancestor of qq
        (rewind q value)
        (if (wind-frame? (car q)) ;evaluate postludes from q
            ((wind-frame-postlude (car q))
             (extend (cdr q)
                     (make-frame
                      (lambda (ignore-q ignore-postlude-value)
                        (unwind (cdr q) value) ) ) ) )
            (unwind (cdr q) value) ) ) )
  (function qq unwind) )
```

Figure 7: `new-call/cc`

Many problems are raised by the introduction of `dynamic-wind`. Not to mention the relationship it carries with partial continuations nor its various possible implementations, it first makes Scheme a purely sequential language with only one possible control point and thus somehow negates the parallel extensions proposed for Scheme. Second, it involves a change in the definition of `call/cc` that seems to preclude to program the old `call/cc` with the new one. Third, it makes the formal semantics of Scheme bigger, more complex and, hardly fitting on a single page since the semantics of `call/cc` grows from a one-liner to half a page.

10 Conclusions

Leaving `dynamic-wind` apart, the four other sets of control operators can be classified as follows. The first two have a dynamic behavior negated by the last two which allow to specify the point up to which reification is wanted: `spawn` and `splitter` make easy to appropriately pair `control`-like with `reset`-like operators. `reset` and `spawn` incorporate, in the reified partial continuation and as last frame, the frame up to which they were reified. This makes it easy to create a new partial continuation from within a partial continuation invocation. `spawn` and `splitter` differ in the amount of reified context as well as in the concept of dynamic extent. It is also unclear what is the "natural" point targeted by a `call/pc`-like operator if this operator is called from within an applied partial continuation and out of the dynamic extent of the original `spawn`-like form.

We can now defend two implementation choices: classes and ACPS. Continuations are completely opaque values in Scheme and, besides applying them, it is not possible to inspect them to know if they contain a given frame or continuation tail. We thus had to make this information explicit and therefore could not use a `call/cc`-based implementation. ACPS, though highly theoretically grounded, is a wonderful tool for that goal and has the additional benefit to be "close" to the implementation i.e., to suggest that continuations are made of frames. Reflection-addicts [dRS, FW84, Wan86, dR87, WF88, DM88] will also appreciate ACPS which allows to represent continuations as list of frames thus opening new opportunities to dissect them.

Classes are useful to factorize the behavior of frames in the `resume` function as well as to allow various control operators to cohabit without interference i.e., without confusing a reset-frame with a wind-frame for instance. Rather than choosing one set of control operators, one may better experiment with all of them simultaneously. In particular, the meaning of `dynamic-wind` with respect to these control operators is obscure and probably deserve more studies. This also the case of the dynamic extent concept.

We hope that these implementations will allow readers to understand control operators better, try to program with them and appreciate their subtleties.

Acknowledgements

Many thanks for all the readers of the many drafts of this paper and especially among them, Matthias Felleisen, Luc Moreau and Amr Sabry.

Bibliography

[App92] Andrew Appel. *Compiling with continuations.* Cambridge Press, 1992.

[CHO88] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, page 124 131, August 1988.

[CR91] William Clinger and Jonathan A Rees. The revised^4 report on the algorithmic language scheme. *Lisp Pointer*, 4(3), 1991.

[Dan89] Olivier Danvy. On listing list prefixes. *Lisp Pointers*, 2(3-4):42–46, January 1989.

[DF89] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Report 89/12, DIKU, DIKU, University of Copenhagen (Denmark), August 1989.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 151–160, Nice (France), June 1990.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. pages 361–391, December 1992.

[DL92] Olivier Danvy and Julia Lawall. Back to direct style II: First-class continuations. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 299–310, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[DM88] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In *LFP '88 – ACM Symposium on Lisp and Functional Programming*, pages 327–341, Snowbird, Utah, July 1988.

[dR87] Jim des Rivières. Control-related meta-level facilities in lisp. In P. Maes and D. Nardi, editors, *Workshop on Meta-Level Architecture and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

[dRS] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *1984 ACM Conference on Lisp and Functional Programming*, pages 331–347.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL '88 – Fifteenth Annual ACM symposium on Principles of Programming Languages*, pages 180–190, San Diego (California USA), January 1988.

[FF87] Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. *Parallel Architectures and Languages Europe*, 259:206–223, 1987.

[FFDM87] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Computer Science Dept. Technical Report 216, Indiana University, Bloomington, Indiana, February 1987.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. In pldi93 [pld93], pages 237–247. Also SIGPLAN Notices 28(6), June 1993.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348–355, Austin, TX., August 1984.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling functional jumps. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[FWH92] Daniel P Friedman, Mitchell Wand, and Christopher Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge MA and McGraw-Hill, 1992.

[HD90] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *PPOPP '90 – ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 128–136, Seattle (Washington US), March 1990.

[HF84] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 18–24, Austin, TX., 1984.

[HF87] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, October 1987.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, TX., 1984.

[IM89] Takayasu Ito and Manabu Matsui. A parallel lisp language PaiLisp and its kernel specification. In Takayasu Ito and Robert H Halstead, Jr., editors, *Proceedings of the US/Japan Workshop on Parallel Lisp*, volume Lecture Notes in Computer Science 441, pages 58–100, Sendai (Japan), June 1989. Springer-Verlag.

[JD88] G F Johnson and D Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *POPL '88 – Fifteenth Annual ACM symposium on Principles of Programming Languages*, pages 158–168, San Diego (California USA), January 1988.

[JG89] Pierre Jouvelot and David K Gifford. Reasoning about continuations with control effects. In *ACM SIGPLAN Programming Languages Design and Implementation*, volume 24 of *SIGPLAN Notices*, pages 218–225, Portland (OR), June 1989. SIGPLAN, ACM Press.

[Joh87] Gregory F. Johnson. Gl – a denotational testbed with continuations and partial continuations. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 165–176, Saint-Paul (Minnesotta USA), June 1987.

[Mat92] Luis Mateu. Efficient implementation of coroutines. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 230–247, Saint-Malo (France), September 1992. Springer-Verlag.

[MH90] Thanasis Mitsolides and Malcolm Harrison. Generators and the replicator control structure in the parallel environment of alloy. In *PLDI '90 –ACM SIGPLAN Programming Languages Design and Implementation*, pages 189–196, White Plains (New-York USA), 1990.

[Mor92] Luc Moreau. An operational semantics for a parallel functional language with continuations. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 415–430, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[MQ93] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. Research report LIX RR 93.05, Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, November 1993.

[pld93] *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque (New Mexico USA), June 1993. ACM Press. Also SIGPLAN Notices 28(6), June 1993.

[QD93] Christian Queinnec and David De Roure. Design of a concurrent and distributed language. In Robert H Halstead Jr and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages 234–259, Boston (Massachussetts USA), October 1993.

[QS91] Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 174–184, Orlando (Florida USA), January 1991.

[Que92a] Christian Queinnec. A concurrent and distributed extension to scheme. In D. Etiemble and J-C. Syre, editors, *PARLE '92 – Parallel Architectures and Languages Europe*, pages 431–446, Paris (France), June 1992. Lecture Notes in Computer Science 605, Springer-Verlag.

[Que92b] Christian Queinnec. Value transforming style. Research Report LIX RR 92/07, Laboratoire d'Informatique de l'École Polytechnique, 91128 Palaiseau Cedex, France, May 1992.

[Que93] Christian Queinnec. Continuation conscious compilation. *Lisp Pointers*, 6(1):2–14, January 1993.

[Rey93] J C Reynolds. The discoveries of continuations. *International journal on Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[SF90a] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation: An International Journal*, 3(1):67–99, January 1990.

[SF90b] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations ii: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programmming*, pages 161–175, Nice (France), June 1990. ACM Press.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about continuation-passing style programs. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 288–298, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

[Sit93] Dorai Sitaram. Handling control. In pldi93 [pld93], pages 147–155. Also SIGPLAN Notices 28(6), June 1993.

[Ste78] Guy Lewis Steele Jr. Rabbit: a compiler for scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[SW74] C Strachey and C P Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monography PRG-11, Oxford University, Computing Laboratory, Oxford University, England, 1974.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. The Lisp Conference, 1980.

[Wan86] Mitchell Wand. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Symposium on LISP and Functional Programming*, pages 298–307, August 1986.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988.