Value Transforming Style*

Research Report LIX RR 92/07, École Polytechnique, May 1992, France.

Christian Queinnec†
École Polytechnique & INRIA-Rocquencourt

Abstract

A new program transformation is presented that allows to remove control operators related to partial continuations. The basis for the transformation is to adopt an improved representation for continuations that makes frames apparent. Various examples of control operators with or without dynamic extent stress are presented.

Scheme [IEE91], offers first-class continuations with indefinite extent. Pioneered in [Lan65, Rey72], continuations proved to be very useful tools allowing to program a wide variety of control features such as coroutines [Wan80], engines [HF84], escapes [HFW84] etc. The `call/cc` function reifies [Wan86] continuations into monadic functions, transfer of control occurs when these functional objects are applied. A transformation called "Continuation Passing Style" (or CPS for short) exists that transforms a program using `call/cc` into a new semantically equivalent program without `call/cc`. In this transformation, continuations are transformed again into monadic functions. CPS has been extensively studied, see [DL92, SF92] for recent developments.

CPS is also a programming style. A function written in CPS-style takes an extra argument known (mis!?)fortunately as the continuation. This continuation represents the final receiver of the value computed by the function. This extra argument does not need to be a real continuation, an object built by `call/cc`, but can simply be a *value transformer* i.e.a regular monadic function without control effect.

Consider for instance the CPS-transformed factorial function:

```
(define (cps-fact n k)
  (if (= n 1) (k 1)
      (cps-fact (- n 1) (lambda (r) (k (* n r)))) ) )
```

One generally invokes this CPS-style `cps-fact` function as `(cps-fact 5 (lambda (x) x))` rather than as `(call/cc (lambda (k) (cps-fact 5 k)))`. This turns out to be psychologically important: people tend to view the `k` variable of `cps-fact` as holding a *value transformer* which is applied on the final result rather than as an escape to the calling point of `cps-fact` with the final value.

Partial continuations are a new concept and allow to abstract over control. Several sets of control operators were proposed in [FWFD88, DF90, QS91]: they emphasize different aspects and have various properties. Unlike regular continuations, partial continuations are real functions that return values: they thus can be composed and resemble more to value transformers. For instance and with the notations of [QS91], the following `factn` function, given a number n, returns a function which will multiply its unique argument by the factorial of n:

```
(define (finalizing-fact n terminator)
  (if (= n 1) (terminator 1)
      (* n (finalizing-fact (- n 1) terminator)) ) )
(define (factn n)
```

*A short version of this report appears in the proceedings of the Wokshop on Staic Analysis, Bordeaux (France), 1992.

†Laboratoire d'Informatique de l'École Polytechnique (URA 1437), 91128 Palaiseau Cedex, France – Email: `queinnec@poly.polytechnique.fr` This work has been partially funded by Greco de Programmation.

```
    (splitter (lambda (g)
                (finalizing-fact
                 n (lambda (one)
                     (call/pc g (lambda (f) (abort g (lambda () f)))) ) ) )) )
  ((factn 4) 2)  →  48
```

A brief explanation of the previous example is: `splitter` labels the current continuation as `g`, `call/pc` reifies the continuation upto `g` into `f`, finally `abort` aborts the computation upto `g` and returns the partial continuation `f`. The `(factn 4)` value is then equivalent to $\lambda x.(4*3*2*x)$ and well represents how a value, injected where `(terminator 1)` appears in the body of `finalizing-fact`, will be transformed into a value restituted by `finalizing-fact`.

An interesting question, raised by Olivier Danvy, was to specify a CPS-like transformation that removes the `splitter`, `call/pc` and `abort` control operators, as defined in [QS91]. An analogous transformation is proposed in [DF90] for the `shift` and `reset` control operators but we cannot reuse this technique since our control operators strongly stress dynamic extent (with run-time checks).

The transformation we will present below, is based on a non classical representation of continuations, known as abstract continuations [FWFD88]. Instead of representing a continuation by an opaque monadic function which can only be applied, it is represented by a list of, say, transformers. To return a value to a continuation is no more a regular invocation, instead the first transformer that appears in the continuation is applied on the rest of the transformers and on the value itself. From an implementer's point of view, continuations have long been associated to stacks, stacks are made of (linked) frames, and frames just wait for values then perform some computations before returning transformed values to the frames below. Transformers thus appear as the abstractions of the frames that populate a stack.

This richer structure for continuations with explicit frames easily allows to encode control operators and particularly ours as well as these of COMMON LISP: `catch`, `throw` and `unwind-protect`. COMMON LISP's dynamic binding can also be explained with frames. Our transformation is nicknamed "value transforming style", or VTS for short. Its interest is to make frames explicit and thus provides a technique to convert a program using partial continuations into a regular program. The transformation may appear as a shift of the semantical equations defining `splitter` *et alii* from the denotational realm to the programmatic world. The transformation is in a sense even better since it removes the side-effect controlling the dynamic extent of `splitter` that was present in the denotation and in the implementation given in [QS91]. This gain is due to a more abstract representation of the continuation in terms of a list of frames.

The paper presents the VTS transformation in section 1 then section 2 explains how to define our control operators on top of it. Other examples such as dynamic binding (section 3), protection (section 4) and the `wind` operator of some Scheme implementations (section 5) are discussed. Related works and conclusion follow.

The paper does not make use of greek letters but rather presents regular Scheme code for two reasons. First, the reification of frames can be given an object-oriented flavor which makes them easy to extend: one can simply program a debugger working on these frames. Second, non-trivial handling of frames, as done by partial continuations, is better expressed in a real programming language.

The paper improves the work on abstract continuations of [FWFD88] and shows that *(i)* abstract continuation can be used as a program transformation rather than a denotational representation of continuations, *(ii)* that control operators on partial continuations like `splitter`, `abort` and `call/pc` but also dynamic binding, protection *à la* `unwind-protect` and `wind` can be given an explicit and functional definition in this program transformation, *(iii)* abstract continuations can be implemented by object technology.

1 Value Transforming Style

The basis of the transformation is to make continuations appear explicitly under a suitable form. Continuations are represented by lists of frames. A continuation is sent a value by means of the `resume` function. A frame is built out of a value transformer by means of `make-frame`. The value transformer can be extracted back from a frame with `frame-transformer`. Finally `extend-q` builds a new continuation from an existing one and a frame. They are defined as follows[1]:

```
(define (resume q value)                              ; Cont×Val→Ans
  (((frame-transformer (car q)) (cdr q)) value) )
(define (extend-q q frame)                            ; Cont×Frame→Cont
  (cons frame q) )
(define (make-frame tr)                               ; Transf→Frame
  (list 'frame tr) )
(define (frame-transformer frame)                     ; Frame→Transf
  (cadr frame) )
```

A value transformer looks like a VTS-transformed monadic function i.e.has the following signature: **Transf = Cont → Val → Ans**. With this representation, the dynamic link is represented by the list spine and is not part of the frame. This will allow to copy frames as required by `call/pc`.

$\mathcal{VTS}[\![\nu]\!]q$	$\rightarrow$	$\texttt{(resume } q\ \nu\texttt{)}$
$\mathcal{VTS}[\![\texttt{(quote } \varepsilon\texttt{)}]\!]q$	$\rightarrow$	$\texttt{(resume } q\ \texttt{(quote } \varepsilon\texttt{))}$
$\mathcal{VTS}[\![\texttt{(if } \pi_0\ \pi_1\ \pi_2\texttt{)}]\!]q$	$\rightarrow$	$\mathcal{VTS}[\![\pi_0]\!]\texttt{(extend-q } q\ \texttt{(make-frame (}\lambda\texttt{(}q'\texttt{)(}\lambda\texttt{(}v'\texttt{) (if } v'\ \mathcal{VTS}[\![\pi_1]\!]q'\ \mathcal{VTS}[\![\pi_2]\!]q'\texttt{)))))}$
$\mathcal{VTS}[\![\texttt{(set! } \nu\ \pi\texttt{)}]\!]q$	$\rightarrow$	$\mathcal{VTS}[\![\pi]\!]\texttt{(extend-q } q\ \texttt{(make-frame (}\lambda\texttt{(}q'\texttt{)(}\lambda\texttt{(}v'\texttt{) (resume } q'\ \texttt{(set! } \nu\ v'\texttt{))))))}$
$\mathcal{VTS}[\![\texttt{(}\lambda\texttt{(}v\texttt{) } \pi\texttt{)}]\!]q$	$\rightarrow$	$\texttt{(resume } q\ \texttt{(}\lambda\texttt{(}q'\texttt{)(}\lambda\texttt{(}v\texttt{) } \mathcal{VTS}[\![\pi]\!]q'\texttt{)))}$
$\mathcal{VTS}[\![\texttt{(}\lambda\texttt{(}v^*\texttt{) } \pi\texttt{)}]\!]q$	$\rightarrow$	$\texttt{(resume } q\ \texttt{(}\lambda\texttt{(}q'\texttt{)(}\lambda\texttt{(}v^*\texttt{) } \mathcal{VTS}[\![\pi]\!]q'\texttt{)))}$
$\mathcal{VTS}[\![\texttt{(}\pi_1\ \pi_2\texttt{)}]\!]q$	$\rightarrow$	$\mathcal{VTS}[\![\pi_1]\!]\texttt{(extend-q } q\ \texttt{(make-frame}$ $\texttt{(}\lambda\texttt{(}q_1\texttt{)(}\lambda\texttt{(}v_1\texttt{) } \mathcal{VTS}[\![\pi_2]\!]\texttt{(extend-q } q_1\ \texttt{(make-frame}$ $\texttt{(}\lambda\texttt{(}q_2\texttt{)(}\lambda\texttt{(}v_2\texttt{) ((}v_1\ q_2\texttt{) } v_2\texttt{)))))))))}$
$\mathcal{VTS}[\![\texttt{(}\pi_1 \ldots \pi_n\texttt{)}]\!]q_0$	$\rightarrow$	$\mathcal{VTS}[\![\pi_1]\!]\texttt{(extend-q } q_0\ \texttt{(make-frame (}\lambda\texttt{(}q_1\texttt{)(}\lambda\texttt{(}v_1\texttt{) } \ldots$ $\mathcal{VTS}[\![\pi_n]\!]\texttt{(extend-q } q_{n-1}\ \texttt{(make-frame (}\lambda\texttt{(}q_n\texttt{)(}\lambda\texttt{(}v_n\texttt{) ((}v_1\ q_n\texttt{) } v_2 \ldots v_n\texttt{)))))))))}$

Table 1: VTS rules

The essence of the VTS transformation appears in table 1. To make it readable, we also include the rules for monadic functions and monadic applications. For VTS to work, any primitive function has to be offered with a new interface. For instance, the old `car` function has to be provided as: `(lambda (q) (lambda (pair) (resume q (car pair))))`. Instead of adding an extra variable to hold the continuation, VTS-transformed abstractions are curryfied with respect to this continuation. This will make easy to simplify administrative redexes. VTS and CPS are different although CPS can be simply obtained from VTS if `extend-q` is defined as `(lambda (q frame) ((frame-transformer frame) q))`.

The VTS transformation tends to produce very huge terms. For instance the simple expression `(car ((lambda (a) '(1)) 2))` is transformed into:

```
VTS[[(CAR ((LAMBDA (A) (QUOTE (1))) 2))]]q
→ (RESUME
    (EXTEND-Q q
     (MAKE-FRAME
      (LAMBDA (Q62)
        (LAMBDA (Q60)
          (RESUME (EXTEND-Q (EXTEND-Q Q62
                             (MAKE-FRAME
                              (LAMBDA (Q63)
                                (LAMBDA (Q61)
                                  (RESUME Q63 (Q60 Q61)) ) ) ) ) )
                   (MAKE-FRAME
                    (LAMBDA (Q66)
                      (LAMBDA (Q64)
                        (RESUME (EXTEND-Q Q66
                                 (MAKE-FRAME
                                  (LAMBDA (Q67)
```

[1] Instead of `k`, we take `q` to name qontinuation. This looks as strange as kontinuation.

```
                                  (LAMBDA (Q65)
                                    ((Q64 Q67) Q65) ) ) ) )
                              2 ) ) ) ) )
                 (LAMBDA (Q68)
                   (LAMBDA (A) (RESUME Q68 '(1))) ) ) ) ) ) )
   CAR )
```

From now, we will simplify expressions such as `(resume (extend-q` q `(make-frame` tr`))` v`)` into `((`tr q`)` v`)` as well as other administrative redexes provided they do not interfere with mutable variables (i.e. v is immutable) otherwise these simplifications would alter the order of evaluation. This is not a problem since mutable variables can be statically determined as well as global variables. These improvements make the VTS-style factorial more readable:

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

The link with CPS now clearly appears. In absence of control operators, the continuation `Q73` will be finally bound to the `Q81` variable when `(EXTEND-Q Q73 ...)` is `resume`-d. The continuation `(EXTEND-Q Q73 (MAKE-FRAME (LAMBDA (Q81) (LAMBDA (V78) ...))))` thus looks like a delayed redex equivalent to `(LAMBDA (V78) (RESUME Q73 (* N V78)))` which is the term CPS would have produced (but for the presence of `RESUME`). The whole interest of VTS lies in this delayed representation which allows to manipulate the individual frames composing the continuation.

As already known, the CPS-equivalent of the `fact` function, `cps-fact`, is strictly more powerful than the original factorial since it allows to receive any monadic function as `k` and, in particular, functions with control effects. This inequivalence also shows when using VTS: the VTS version of `cps-fact` is not equal to the VTS transform of `fact`.

We take care to represent frames as instances of an abstract data type. We can now introduce new kinds of frames with new properties but still respecting this interface. Lists can be banished in favor of λ-terms emulating `cons`, `car` and `cdr`. Another convenient style is to adopt an object-oriented view[2] and to consider `frame` as a class with one slot: `transformer`:

```
(define-class frame (Object) transformer)
;;; defines make-frame, frame? and frame-transformer
```

We will adopt this view for the rest of the paper.

2 Partial continuation operators

This section sketches how to encode our control operators. First of all, we define `group-frame` as a subclass of `frame`. A group-frame enriches a frame with one slot to hold a group. A *group* is the first class object on which `splitter` invokes its argument. This term comes from [Osb90]'s *sponsor* and [HD90]'s *controller*: a group is responsible for a computation and, as suggested by the above references, controls the possible concurrent threads that might be created for this end [Que92][3]. This aspect of groups is uninteresting for our purpose as far as they can be created by `make-group` and compared by means of `group-eq?`.

[2] We adopt a small but efficient object system called Meroon.

[3] These features appear in the Idiom of Icsla, a quasi-Scheme implementation under progress at INRIA and École Polytechnique.

```
(define-class group-frame (frame) group)
;;; defines make-group-frame, group-frame? and group-frame-group
```

The `splitter`[4] operator can now be straightforwardly defined. The group-frame is simply "pushed" on the continuation, its associated value transformer just pops it. User invokable functions are prefixed by `VTS-` and adopt the interface conventions specified by VTS. Non prefixed functions belong to the implementation.

```
(define VTS-splitter
  (lambda (q)
    (lambda (function)
      (let ((group (make-group)))
        ((function (extend-q q (make-group-frame
                                (lambda (q) (lambda (value) (resume q value)))
                                group )))
         group ) ) ) ) )
```

The `in-group?` predicate allows to check dynamic extent constraints. The `in-group?` predicate is an internal utility function that will be used below. It can be turned into a user accessible predicate, see `within/de?`[5] definition. This predicate returns false if the invoker is not in the dynamic extent of the specified group. Otherwise it will return a dotted pair containing the original continuation cut into two pieces: the part strictly above the group-frame and the part below. Since more than one group-frame can appear[HD90, end of section 3], the deepest is preferred. This function is for explanatory purpose, more efficient functions can be easily devised. A more object-oriented flavor can be also proposed with generic functions.

```
(define (in-group? group q)
  (define (searched-group? group frame)
    (cond ((group-frame? frame) (group-eq? group (group-frame-group frame)))
          ((frame? frame) #f) ) )
  (define (look q qq)
    (and (pair? q)
         (if (searched-group? group (car q))
             (or (look (cdr q) (cons (car q) qq))
                 (cons (reverse qq) q) )
             (look (cdr q) (cons (car q) qq)) ) ) )
  (look q '()) )

(define VTS-within/de?
  (lambda (q)
    (lambda (group)
      (resume q (not (not (in-group? group q)))) ) ) )
```

The `abort` operator replaces the current computation upto a specified group by a new computation which is performed under the control of the same group. `abort` can only be used while in the dynamic extent of the specified group. `abort` is simply written as:

```
(define VTS-abort
  (lambda (q)
    (lambda (group thunk)
      (let ((above+below (in-group? group q)))
        (if above+below ((thunk (cdr above+below)))
                        (wrong "out of extent") ) ) ) ) )
```

The last control operator `call/pc` reifies the partial continuation upto the specified group provided this invocation is made while in the dynamic extent of the group.

[4] Also named `call/de` for "call with dynamic extent" in the Idiom of Icsla.
[5] This name stands for "within dynamic extent".

```
(define VTS-call/pc
  (lambda (q)
    (lambda (group function)
      (let ((above+below (in-group? group q)))
        (if above+below ((function q)
                          (lambda (qq)       ← the reified partial continuation
                            (lambda (value)
                         (resume (append (car above+below) qq) value) ) ) )
            (wrong "out of extent") ) ) ) ) )
```

This ends the definition of our control operators. The VTS transformation thus converts a program using `splitter`, `abort` and `call/pc` in a new equivalent program using `VTS-splitter`, `VTS-abort` and `VTS-call/pc` which are entirely functionally defined. A consequence is that Scheme's `call/cc` can be defined by VTS since it is definable in terms of the previous control operators [QS91].

Examples

The result of performing VTS on `factn` and `finalizing-fact` is:

```
(DEFINE FINALIZING-FACT (LAMBDA (Q81)
  (LAMBDA (N TERMINATOR)
    (IF (= N 1) ((TERMINATOR Q81) 1)
        ((FINALIZING-FACT (EXTEND-Q Q81
                            (MAKE-FRAME
                             (LAMBDA (Q93)
                               (LAMBDA (V90)
                                 (RESUME Q93 (* N V90)) ) ) ) ) ))
         (- N 1)
         TERMINATOR ) ) ) ))

(DEFINE FACTN (LAMBDA (Q21)
  (LAMBDA (N)
    ((VTS-SPLITTER Q21)
     (LAMBDA (Q26)
       (LAMBDA (G)
         ((FINALIZING-FACT Q26)
          N
          (LAMBDA (Q33)
            (LAMBDA (ONE)
              ((VTS-CALL/PC Q33)
               G
               (LAMBDA (Q40)
                 (LAMBDA (F)
                   ((VTS-ABORT Q40)
                    G
                    (LAMBDA (Q47)
                      (LAMBDA ()
                        (RESUME Q47 F) ) ) ) ) ) ) ) ) ) ) ) ) ) ))
```

3 Dynamic binding

This section just discusses a possible implementation of dynamic binding with deep binding. The basic idea is to "push" dynamic-frames onto the continuation that record dynamic associations[6].

The first operator associates two objects while computing `thunk`. The value of `key` is associated to the value of `value`. Note that it is possible for the key to be anything and not just a symbol; a similar behavior

[6] The two following operators are implemented in the Idiom of Icsla.

can be observed for `catch` tags in COMMON LISP. The name `assoc/de` stands for "associate with dynamic extent".

```
(define-class association-frame (frame) key value)

(define VTS-assoc/de
  (lambda (q)
    (lambda (key value thunk)
      ((thunk (extend-q q (make-association-frame
                            (lambda (qq)
                              (lambda (value) (resume qq value)) )
                            key
                            value )))) ) ) )
```

A value can be searched in the dynamic environment by means of the `find/de` operator. It takes two continuations, the `success` continuation will be invoked on the value if found, otherwise the `failure` thunk is invoked. All comparisons are done with `eqv?`.

```
(define VTS-find/de
  (lambda (q)
    (lambda (key success failure)
      (define (find qq)
        (if (pair? qq)
            (cond ((association-frame? (car qq))
                   (if (eqv? key (association-frame-key (car qq)))
                       ((success q) (association-frame-value (car qq)))
                       (find (cdr qq)) ) )
                  (else (find (cdr qq))) )
            ((failure q)) ) )
      (find q) ) ) )
```

The previous definition corresponds to a deep binding implementation where associations are recorded in an A-list spread over the continuation. The main point of interest is that VTS gives a precise and understandable meaning to this kind of binding. Of course other semantics are possible.

4 Protection

The control operators for partial continuations explained in [QS91] emphasize dynamic extent. The `unwind-protect` feature of COMMON LISP, allows to start a computation when some expression returns a value or in case of escape. We propose here to show how such a feature can be added to our set of control operators and defined in terms of VTS. This new operator is named `protect` and is implemented in the Idiom of Icsla. It takes a `thunk` (the expression to evaluate) and a `cleaner` which will be invoked when the thunk returns a value or is `abort`-ed. Since `abort` is called with a thunk, the cleaner will take such a thunk and will transform it into another thunk. Protect-frames can be captured by partial continuations. Let us give some examples of `protect` uses:

```
(protect (lambda () 1)
         (lambda (thunk) (lambda () (* 2 (thunk)))) )      → 2
(splitter (lambda (g)
            (* 2 (protect (lambda ()
                            (call/pc g (lambda (f) ;f = λx.(* 10 x)
                                         (f (f 3)) )) )
                          (lambda (thunk) (lambda () (* 5 (thunk)))) )) ))
  3000
```

To define `protect` is simple: it simply pushes a protect-frame on the continuation. This in turn requires to modify abort to take into account these protect-frames. Observe that the cleaners are invoked with the dynamic bindings that were current at the time they were installed (as in COMMON LISP).

```
(define-class protect-frame (frame) cleaner)

(define VTS-protect
  (lambda (q)
    (lambda (thunk cleaner)
      ((thunk (extend-q q (make-protect-frame
                           (lambda (q)
                             (lambda (value)
                               ((cleaner
                                 (extend-q q (make-frame
                                              (lambda (qq)
                                                (lambda (thunk)
                                                  ((thunk qq)) ) ) )) )
                                (lambda (qq)
                                  (lambda () (resume qq value)) ) ) ) )
                           cleaner )))) ) ) )

(define VTS-abort-with-protect    ; The new abort
  (lambda (q)
    (lambda (group thunk)
      (define (unroll q qq thunk)
        (if (pair? qq)
            (cond ((protect-frame? (car q))
                   (((protect-frame-cleaner (car q))
                     (extend-q (cdr q)
                               (make-frame
                                (lambda (qqq)
                                  (lambda (new-thunk)
                                    ((abort-with-protect qqq)
                                     group new-thunk ) ) ) ) ) )
                    thunk ) )
                  (else (unroll (cdr q) (cdr qq) thunk)) )
            ((thunk q)) ) )
      (let ((above+below (in-group? group q)))
        (if above+below
            (unroll q (car above+below) thunk)
            (wrong "Out of extent") ) ) ) ) )
```

This new feature fits well with the previous ones since these were stressing dynamic extent. Combining protect with within/de? allows to detect the final end of an expression even if it has been captured in some partial continuations. This problem can be illustrated on the call-with-output-file function. The Scheme standard explicitly states that it can be defined as:

```
(define (call-with-output-file filename proc)
  (let ((port (open-output-file filename)))
     (let ((result (proc port)))
        (close-output-port port)
        result ) ) )
```

The intention is that when a file is used in the dynamic extent of call-with-output-file then it is automatically closed when normally returning. This definition has undesirable effects in conjunction with splitter. Consider for instance:

```
(splitter
```

```
(lambda (mark)
  (cons 'a (call-with-output-file
             "file"
             (lambda (out)
               (display 1 out) ;OK
               (call/pc mark (lambda (c)
                               (display 2 out) ;still OK
                               (c 'b)          ;→(a . b)
                               (display 3 out) ;error !
                                )) ) )) ) )
```

If `call-with-output-file` is defined as above then the partial continuation c is *(λ(x) (close-output-port out) x)*, the port is thus closed ! It may be considered as natural there, since invoking c is done in the dynamic extent of `call-with-output-file`, that the port is only closed when returning *the last time* from `call-with-output-file`. This can be programmed with

```
(define (call-with-output-file filename proc)
  (let ((port (open-output-file filename))
        (m 'wait) )
    (protect (lambda () (splitter (lambda (mark)
                                    (set! m mark)
                                    (proc port) ) ) )
             (lambda (thunk)
               (if (within/de? m)
                   thunk
                   (begin (close-output-port port)
                          thunk ) ) ) ) ) )
```

With this definition the previous excerpt is not erroneous.

5 The `wind` operator

Some implementations of Scheme offer a `wind` function which allows to recreate a kind of dynamic environment. It is useful when simulating different processes with a sequential semantics. The meaning of `wind` is particularly unclear and its implementation somewhat obscure [FWH92]. The following definitions for `call/cc` and `wind` are purely functional and more precise.

The `wind` operator pushes wind-frames onto the continuation. These frames will be used by the "continuations" created by `call/cc` in order to run preludes and postludes. Wind-frames are numbered by `make-wind-id` in order to be compared. Note that, in presence of `wind`, `call/cc` creates a rather more complex object than indicated in the pure denotation of Scheme.

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

(define VTS-wind
  (lambda (q)
    (lambda (prelude thunk postlude)
      ((prelude (extend-q (extend-q q
                           (make-wind-frame
                            (lambda (qq)
                              (lambda (value)
                                ((postlude
                                  (extend-q qq
                                   (make-frame
                                    (lambda (qqq)
                                      (lambda (ignore-postlude-value)
                                        (resume qqq value) ) ) ) ) )) ) )
```

```
                        prelude
                        postlude
                        (make-wind-id) ) )
                   (make-frame
                    (lambda (qq)
                      (lambda (ignore-prelude-value)
                        ((thunk qq)) ) ) ) ))) ) ) )

(define VTS-call/cc
  (lambda (q)
    (lambda (function)
      (define (find-wind-qs q result) ;return the reversed list of all suffixes
        (if (pair? q)                 ;of q that begin with a wind-frame
            (cond ((wind-frame? (car q))
                   (find-wind-qs (cdr q) (cons q result)) )
                  (else (find-wind-qs (cdr q) result)) )
            result ) )
      (let* ((q-wind-qs (find-wind-qs q '()))
             (q-ids (map (lambda (q) (wind-frame-id (car q)))
                         q-wind-qs )) )
        (define (rewind qs value)
          (if (pair? qs)
              (((wind-frame-prelude (caar qs))
                (extend-q (cdar qs)
                 (make-frame
                  (lambda (ignore-q)
                    (lambda (ignore-prelude-value)
                      (rewind (cdr qs) value) ) ) ) ) ))
              (resume q value) ) )
        (define (unwind qq qs value)
          (if (pair? qq)
              (cond ((wind-frame? (car qq))
                     (if (member (wind-frame-id (car qq)) q-ids)
                         (rewind qs value)
                         (((wind-frame-postlude (car qq))
                           (extend-q (cdr qq)
                            (make-frame
                             (lambda (qqq)
                               (lambda (ignore-postlude-value)
                                 (unwind qqq qs value) ) ) ) ) )) ) )
                    (else (unwind (cdr qq) qs value)) )
              (rewind qs value) ) )
        ((function q)
         (lambda (qq)        ← the reified full continuation
           (lambda (value)
             (unwind qq q-wind-qs value) ) ) ) ) ) ) )
```

This definition shows well that continuations created by `call/cc` unwinds the continuation executing all postludes then rewinds the continuation upto the target point executing all preludes.

6 Conclusion

The VTS transformation belongs to the family of CPS conversions in that it makes continuations explicit. It differs from CPS since it uses a non functional representation of continuations which allows frames to be explicit. Making frames explicit allows to give a functional definition for a set of control operators dealing with partial continuations and emphasizing dynamic extent. This paper is thus a positive answer to the initial quest for a CPS-like transformation removing these control operators. The VTS transformation is general and can be applied with success to dynamic bindings and other control operators such as `protect` and Scheme's `wind`.

New questions can also be raised after this work. For instance, does the inverse CPS transformation of [DL92], recover from a VTS transformed program, the implementation of `splitter` *et alii* with three `call/cc` ? The VTS transformation does not imply for now the possibility for the user to grab the representation of the continuation: there is no *causal connection* but this may be simply offered. The user may then subclass the `frame` class and thus design new behaviors.

Bibliography

[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.

[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).

[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.

[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.

[IEE91] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language.* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[Lan65] P J Landin. A Correspondence between algol 60 and Church's Lambda-notation. *Communications of the ACM*, 8(?):89–101 and 158–165, February 1965.

[Osb90] Randy B. Osborne. Speculative computation in MultiLisp, an overview. In *LFP '90 – ACM Symposium on Lisp and Functional Programming*, pages 198–208, Nice (France), 1990.

[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.

[Que92] 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.

[Rey72] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[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).

[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.