A Better API for First-Class Continuations

Marc Feeley
Département d'informatique et recherche opérationnelle
Université de Montréal
`http://www.iro.umontreal.ca/~feeley`
(Short paper)

Abstract

Scheme's `call-with-current-continuation` (`call/cc`) procedure reifies the implicit continuation into a procedure. Calling this procedure causes this continuation to be invoked with the argument(s) as result(s). This approach is consistent with the traditional representation of continuations as functions in denotational semantics. This simple interface is restrictive and inefficient. We propose the use of an abstract data type to represent first-class continuations, and motivate the operations it supports with applications in search algorithms, exception handling, and debugging.

1 Introduction

We will motivate the need for a better API for first-class continuations using a few sample programs.

1.1 Search with Exit

First-class continuations are sometimes used in search procedures to abort the rest of the search when the desired item is found. Assume we need to search a binary tree made of pairs with leaves that are non-pairs. We want to find the first leaf x in a left-to-right scan for which `(pred` x`)` is true and return `(process` x`)`, where `pred` and `process` are parameters of the search procedure. The search procedure indicates failure by returning a special value. The following solution uses a recursive `find` procedure which receives a continuation (`abort`) obtained with `call/cc` to allow it to return from the `search` procedure:

```
(define search
  (lambda (tree pred process)
    (call/cc
     (lambda (abort)
       (find tree pred process abort)))))

(define find
  (lambda (tree pred process abort)
    (if (pair? tree)
        (begin
          (find (car tree) pred process abort)
          (find (cdr tree) pred process abort))
        (if (pred tree)
            (abort (process tree))
            'could-not-find-it))))
```

Note that the first recursive call to `find` is not a tail-call which means that when the appropriate leaf is found the current continuation is the continuation of the call to `search` to which has been added one continuation frame per left branch in the path from the root to the leaf.

The code computes the correct result but unfortunately it consumes more space for the continuation than is really required. The problem is that the call `(abort (process tree))` only calls the abort continuation **after** the call `(process tree)` returns, and it is only when the abort continuation is called that the continuation frames created inside `find` will all be reclaimed. This means that the call to `process` is not a tail-call even if it appears to be a tail-call with respect to `search` (in a sense they both have the same "continuation"). This may be a serious problem if a loop is formed by `process` tail-calling `search`.

A known trick to solve this problem, which we will call "pop-and-call", is to wrap an extra pair of parentheses around the `call/cc` call:

```
(define search
  (lambda (tree pred process)
    ((call/cc
      (lambda (abort)
        (find tree pred process abort))))))
```

and to thunkify the argument to the continuation in `find`, i.e. `(abort (lambda () (process tree)))`, and the failure case, i.e. `(lambda () 'could-not-find-it)`. So when the leaf is found the `abort` continuation is immediately invoked, thus reclaiming all the continuation frames, and `process` receives exactly the same continuation as `search`.

The continuations created by a program form a tree, with the primordial continuation of the program at its root. When a procedure is called in non-tail position, the branch of the tree corresponding to the current continuation is extended. When a procedure returns, the current continuation moves to the frame one closer to the root. In this model `call/cc` simply creates a reference to the current continuation, and invoking a continuation moves the current continuation arbitrarily. In this tree metaphor, the operation needed in the search example corresponds to grafting a new branch. We want to move the continuation to a particular point **and then** start a new branch by performing a call.

For the following examples we will abstract these continuation operations with three procedures, using the pop-and-call trick for implementation:

```
(define continuation-capture
  (lambda (receiver)
    ((call/cc (lambda (cont)
                (lambda () (receiver cont)))))))
```

```
(define continuation-graft
  (lambda (cont thunk)
    (cont thunk)))

(define continuation-return
  (lambda (cont . returned-values)
    (continuation-graft
     cont
     (lambda () (apply values returned-values)))))
```

Note that `continuation-return` allows any number of values to be returned to a continuation. Also note that `continuation-capture` tail-calls the `receiver`.

1.2 Exception Handling and Dynamic Environments

The *dynamic environment* is a structure which allows the system to find the value returned by `current-input-port`, `current-output-port`, etc. The procedures `with-input-from-file`, `with-output-to-file`, etc extend the dynamic environment to produce a new dynamic environment which is in effect for the dynamic extent of the call to the thunk passed as the last argument. Some Scheme systems generalize the dynamic environment by providing procedures and special forms to define new *dynamic variables* and bind them in the dynamic environment. For instance, Chez-Scheme [Dyb98], MzScheme [Fla00], Larceny [HC00], Chicken [Win01] and Gambit-C [Fee01a] have the procedure `(make-parameter` *initial-value*`)` which creates a global dynamic variable and returns a procedure to get and set the value of the variable in whichever dynamic environment is current at the time of the call, and the form `(parameterize ((`*variable new-value*`)...)` *body*`)` to bind the *variable* for the dynamic extent of *body*'s evaluation.

Clearly there is an interaction between continuations and dynamic environments. A reasonable model is to conceptually attach to each continuation the dynamic environment which was current when that continuation was created. When the continuation is invoked the current dynamic environment is restored accordingly. The multithreading support SRFI [Fee01b] mandates this model and introduces an exception handling procedure based on it. An internal dynamic variable contains the current exception handler procedure (a procedure that is called when an exception is raised). The procedure call `(with-exception-handler` *handler thunk*`)` binds the exception handler to *handler* for the dynamic extent of the call to *thunk*.

Suppose we need to implement an exception handling mechanism that models the `try-catch-throw` of Java. It needs to capture the continuation of the `try` so that the `catch` that processes a caught exception can be executed with the same continuation as the `try`. If `try`'s continuation was not restored the program would enter an infinite loop if an exception is raised in the `catch`. Because `with-exception-handler` binds the handler during the dynamic extent of the call to the thunk we need to exit this dynamic extent when the exception is processed. The continuation grafting operation can thus be used to implement a Java style exception handling mechanism:

```
(define try-catch
  (lambda (catcher thunk)
    (continuation-capture
     (lambda (cont)
       (with-exception-handler
         (lambda (e) ; e = caught exception
           (continuation-graft cont
                               (lambda () (catcher e))))
         thunk)))))
```

This will not lead to an infinite loop:

```
(try-catch
  (lambda (e)
    (display "caught exception")
    (/ 2 0)) ; second exception raised
  (lambda ()
    (display "computing 1/0")
    (/ 1 0))) ; first exception raised
```

1.3 Debugger Evaluations

The need to perform a call within a specific continuation is also useful for implementing debuggers in Scheme. Suppose that a program encounters a divide-by-zero error in the expression `(+ (/ x (f x)) 1)` and this causes the program to suspend its execution at the point of error and a read-eval-print loop is presented to the user allowing her to evaluate expressions to discover and repair the problem. In this particular case perhaps she wants to know what the value of "`x`" and "`(f x)`" are. It is useful for these evaluation requests to be performed in the context of the error. For one thing we want the lexical environment to be the same so that the appropriate binding of "`x`" and "`f`" can be found. Let's assume this issue is solved. It is also important for the dynamic environment to be the same so that the evaluation of a call to `f` can access the correct current output port, current exception handler, and any user introduced dynamic variable (if the system has a user accessible dynamic variable binding construct, such as "`parameterize`"). The debugger, which has possibly extended or changed the dynamic environment for its own needs (e.g. rebinding the current output port), must return to the dynamic environment at the error point, to evaluate the user supplied expression, and then return to the dynamic environment of the debugger, for printing the result. Continuation grafting can once again be used to implement this:

```
(define /
  (lambda (a b)
    (if (= b 0)
        (debugger "divide-by-zero")
        (* a (expt b -1)))))

(define debugger
  (lambda (error-msg)
    (continuation-capture
     (lambda (there)
       (with-input-from-port
         (get-standard-input-port)
         (lambda ()
           (with-output-to-port
             (get-standard-output-port)
             (lambda ()
               (display error-msg)
               (let loop ()
                 (display "> ")
                 (let ((x (read))
                       (e (continuation-environment there)))
                   (write
                    (continuation-capture
                     (lambda (back)
                       (continuation-graft
                        there
                        (lambda ()
                          (continuation-return
                           back
                           (eval x e)))))))
                   (loop)))))))))))
```

Note that it is important to use `continuation-return` for returning to `back` so that `eval` is called in the correct dynamic environment. Moreover, to pass the correct lexical environment to `eval`, we have assumed the existence of the procedure `(continuation-environment` *cont*`)` which extracts the lexical environment of a continuation *cont* obtained with `continuation-capture`. To allow optimizations by the compiler this procedure may have to return a partial environment (the precise semantics are beyond the scope of this paper).

1.4 Debugger Continuation-Crawling

The pop-and-call trick is unfortunately not very modular. The piece of code that captures the continuation and the sites that invoke the continuation must know that the continuation is invoked in a special way. This problem is minor in the above applications, but it is more severe when we consider the previous debugger extended with the ability to examine each frame in the continuation (to know the nesting of the procedures that led to the error) and perform an evaluation in any of these continuation frames.

A new procedure is needed to walk up the continuation, for instance we could define `(continuation-next` *cont*`)` to take a continuation *cont* obtained with `continuation-capture` and return it with the top-most frame removed, or `#f` if it only contains a single frame. Such a procedure can't be implemented with the pop-and-call trick because there is no way to identify all possible points in the program (and run-time library) that create continuation frames and insert a call to `continuation-capture`.

2 Continuations Should be an Abstract Data Type

The procedure representation of continuations can only support a single operation: returning value(s) to the continuation. To allow grafting and possibly other operations on continuations (such as `continuation-environment` and `continuation-next` and finding the procedure that created a continuation for reporting the location of an error, etc) it makes sense to introduce a distinct data type for continuations and a set of operations on them. Not only is this more flexible but it can be implemented more efficiently than the procedure representation. The data type could be exactly the continuation objects manipulated by the system thus avoiding the space and time overheads of wrapping these objects in closures.

Our proposal, which we plan to submit as a SRFI, will contain at its core this set of primitives on the continuation data type:

- `(continuation-capture` *receiver*`)` – creates a continuation object representing the current continuation and tail-calls the *receiver* procedure with this continuation as the single argument.
- `(continuation-graft` *cont thunk*`)` – calls the *thunk* with no argument and the implicit continuation *cont*.
- `(continuation-return` *cont value*1`...)` – returns the value(s) to the continuation *cont* (the definition given above in terms of `continuation-graft` is still valid).

Note that `call/cc` can easily be implemented on top of these primitives:

```
(define call/cc
  (lambda (receiver)
    (continuation-capture
     (lambda (k)
       (receiver (lambda returned-values
                   (apply continuation-return
                          k
                          returned-values)))))))
```

Of course the pop-and-call trick allows these primitives to be simulated using `call/cc`, until introspective primitives such as `continuation-next` are made available.

The main interest of these primitives in the short term will be higher efficiency of continuation operations for systems that implement them natively (i.e. not with pop-and-call). In addition to the space savings of continuation grafting, the primitives avoid the creation of a closure by `call/cc` and the indirection through this closure when the continuation is restored. For the same reasons, it might make it easier to teach first-class continuations with these primitives (students are often lost in the tangle of procedures that are involved when `call/cc` is used).

We also expect that the existence of these primitives will pave the road for a wider range of introspective extensions to Scheme, for example a SRFI with the features necessary for implementing a portable debugger in Scheme or for composing continuations.

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[Dyb98] R. Kent Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, Bloomington, Indiana, 1998.

[Fee01a] Marc Feeley. Gambit-C version 4.0. To appear at `http://www.iro.umontreal.ca/~gambit`, 2001.

[Fee01b] Marc Feeley. SRFI-18: Multithreading support. Available at `http://srfi.schemers.org/-srfi-18/srfi-18.html`, March 2001.

[Fla00] Matthew Flatt. PLT MzScheme: Language Manual. Available at `http://www.cs.rice.edu/CS/-PLT/packages/doc/mzscheme/`, August 2000.

[HC00] Lars T. Hansen and William D. Clinger. Larceny User's Manual. Available at `http://-www.ccs.neu.edu/home/lth/larceny/manual/`, September 2000.

[Win01] Felix L. Winkelmann. CHICKEN: A simple and portable Scheme system - User's manual. Available at `http://www.call-with-current-cont-inuation.org/manual.html`, 2001.