Call/cc and continuations

Ossi Vääränen

ossi@niksula.hut.fi
Call/cc: definition

Call/cc is a unary function that calls its argument, which also must be a unary function, with something called the current continuation. The current continuation in turn is an abstraction of the rest of the program.
Note: the “real” name for call/cc is actually call-with-current-continuation, but call/cc is a semi-standard way to call it.
Example:

\[(+ 1 \text{(call/cc (lambda (k) (+ 2 (k 3)))})\)

In the code above, \((+ 1 [\ ]\) represents the whole program’s continuation to call/cc when it is invoked.

The point with call/cc here is that the current continuation, or the future evaluation of the program, \((+ 1 [\ ]\), is stored in \(k\). When \(k\) is then called in call/cc’s argument function, the argument function abandons its own computation and replaces it with \(k\)!
> (+ 1 (call/cc
   (lambda (k) (+ 2 (k 3))))))

⇒ (+ 1 3)
4

This kind of continuation is often referred to as an *escaping* continuation.
Another example:

We create \texttt{r}...

\begin{verbatim}
> (define r #f)
\end{verbatim}

...to store the continuation into \texttt{r}:

\begin{verbatim}
> (+ 1 (call/cc (lambda (k)
  (set! r k)
  (+ 2 (k 3)))))
\end{verbatim}

\texttt{4}

\begin{verbatim}
> (r 4)
\end{verbatim}

\texttt{5}
What happened?

> (define r #f)
> (+ 1 (call/cc (lambda (k)
>                        (set! r k)
>                        (+ 2 (k 3))))))

In the argument function of call/cc above, the continuation passed to the function is stored to \(r\). Thus it is globally (and hence permanently) available in \(r\), evaluating to \((+ 1 [])\).
More on escaping continuations...

Consider the following code:

```
(define (list-product s)
  (let recur ((s s))
    (if (null? s) 1
        (* (car s) (recur (cdr s))))))
```

The code above calculates the product of all elements in a given list. If, however, any element of the list is zero, the code still traverses the rest of the list, and thus wasting many valuable bits.
A continuation-based approach:

(define (list-product-c s)
  (call/cc
   (lambda (exit)
     (let recur ((s s))
       (if (null? s) 1
         (if (= (car s) 0) (exit 0)
           (* (car s) (recur (cdr s)))))))))))
Tree matching: a canonical example

Consider the problem of determining whether two trees have the same fringe, e.g. their leaves, when accessed in a left-to-right order, equal to each other.

A functional approach to the problem would be to devise a procedure that flattens the tree and then tests each corresponding pair of elements from both flattened lists.
(define same-fringe?
  (lambda (tree1 tree2)
    (let loop ((ftree1 (flatten tree1))
               (ftree2 (flatten tree2)))
      (cond ((and (null? ftree1) (null? ftree2)) #t)
            ((or (null? ftree1) (null? ftree2)) #f)
            ((eqv? (car ftree1) (car ftree2))
             (loop (cdr ftree1) (cdr ftree2)))
            (else #f)))))

(define flatten
  (lambda (tree)
    (cond ((null? tree) '())
          ((pair? (car tree))
           (append (flatten (car tree))
                   (flatten (cdr tree))))
          (else
           (cons (car tree)
                 (flatten (cdr tree)))))))
> (same-fringe? '(1 2) 3) '(1 (2 3)))
#t
This solution is not a terribly efficient one, although simple and comprehensible. Note that each call to the loop construct calls \texttt{flatten} for both trees, which causes a lot of consing and thus wastes memory a lot (?).

The \texttt{same-\textbf{fringe}\texttt{?}} procedure can be as well implemented using continuations. This way we can avoid any \texttt{cons}ing that happened in the previous approach. We define a procedure \texttt{tree-\textgreater generator}, which takes a tree and returns a procedure that returns the leaves of the tree, one at a time:
(define tree->generator
  (lambda (tree)
    (let ((caller '*))
      (letrec
        ((generate-leaves
           (lambda ()
             (let loop ((tree tree))
               (cond ((null? tree) 'skip)
                     ((pair? tree)
                      (loop (car tree))
                      (loop (cdr tree)))
                     (else
                      (call/cc
                       (lambda (rest-of-tree)
                        (set! generate-leaves
                           (lambda ()
                             (rest-of-tree 'resume))
                           (caller tree)))))))))
        (caller '()))))
  (lambda ()
    (call/cc
     (lambda (k)
      (set! caller k)
      (generate-leaves))))))
The tree-matching procedure looks suspiciously simple:

\[
\text{(define same-fringe?}
\begin{align*}
\text{(lambda (tree1 tree2))} \\
\text{(let ((gen1 (tree->generator tree1)))} \\
\text{(let ((gen2 (tree->generator tree2)))} \\
\text{(let loop ()} \\
\text{(let ((leaf1 (gen1)))} \\
\text{(let ((leaf2 (gen2)))} \\
\text{(if (eqv? leaf1 leaf2) } \\
\text{(if (null? leaf1) #t (loop))} \\
\text{#f))})))))
\end{align*}
\]
Catch/throw

Catch/throw is a construct in which a body of expressions can be enclosed in a `catch` block. A `catch` expression takes a tag, which can be any kind of object, followed by a body of expressions. At any point within this code, or code called by it, a `throw` with a corresponding tag will cause the `catch` expression to return immediately.
Welcome to Macintosh Common Lisp Version 4.3!

? (defun sub ()
   (throw 'abort 42))
SUB

? (defun super ()
   (catch 'abort
      (sub)
      (format t "We’ll never see this.")
    ))
SUPER

? (super)
42
Catch/throw: a Scheme implementation

First, let’s describe what catch and throw do:

**catch** does two things: First, it must know how to continue if a throw to it is made. Thus, catch must save its current continuation. Second, it must be accessible and identifiable by its tag by the throw that may look after it.

**throw** looks for a catch expression with a certain tag, and gives control to it.
First we implement code to store the catch information globally. For each catch we need a tag for identification and a continuation for the throw to work:
(define tags '()) ; init taglist

(define (add-tag! name cont)
  (let ((val (assq name tags)))
    (if (not val)
      (set! tags
        (cons (list name cont)
              tags))
      (set-cdr! val
        (cons cont (cdr val))))))

(define (remove-tag! name)
  (define (iter lst)
    (if (eqv? (caar lst) name)
      (if (null? (cddar lst))
        (cdr lst)
        (cons (cons (caar lst) (cddar lst)) (cdr lst)))
      (cons (car lst) (iter (cdr lst))))
    (set! tags (iter tags)))
Now we define `catch` as a macro that catches its caller’s current continuation, stores it to the global `tags` list, and starts evaluating its body.
(define-macro catch
  (lambda (tag . body)
    '(let ((c (call/cc (lambda (c)
                    (add-tag! ,tag c)
                    ,@body
                ))))
      (remove-tag! ,tag c)))
)

The call to remove-tag! is necessary, since without it, a catch expression would leave the tag-continuation pair to the global tags list.
Throw is implemented as

(define (throw tag val)
  (let ((t (assq tag tags)))
    (if t
      ((cadr t) val)
      (error "can’t throw---tag not found!!!!1!" name))))

The code looks for tag in the global tags list, and if found, and applies its associated continuation to the throw value, which in turn is equal to letting the catch block return with val.
Indeed,
> (define (sub)
  (throw 'mung 42))
> (define (super)
  (catch 'mung
    (sub)
    (display "We shouldn’t see this.")))
> (super)
42
We can also nest `catch` expressions:
> (define (sub)
   (throw 'mung 42))
> (define (sub2)
   (throw 'barf 666))
> (define (super)
   (catch 'mung
     (catch 'barf
       (sub)
       (display "Shouldn’t come here."))
     (sub2)
     (display "Not here, either.")))
> (super)
42
Coroutines

The previous tree->generator procedure used continuations to store the procedure’s state to an internal variable each time before returning a value to its caller. This enabled the procedure to resume its computation where it was left the last time.

This mechanism can be generalized further to procedures that resume each other. Such procedures are called coroutines.
A coroutine is a unary procedure whose body can contain \texttt{resume} calls. \texttt{resume} is a two-argument procedure used by a coroutine to resume another coroutine with a transfer value.
We define a coroutine macro that handles the resume calls for us:

(define-macro coroutine
  (lambda (x . body)
    `(letrec ((local-control-state
                   (lambda ,(x) ,@body))
             (resume
              (lambda (c v)
                (call/cc
                 (lambda (k)
                   (set! local-control-state k)
                   (c v))))))
    (lambda (v)
      (local-control-state v))))
Now we return to the tree-matching problem. The problem is solved with three coroutines: a matcher coroutine, that \texttt{resumes} the two leaf-generating coroutines, and the leaf generators.
(define make-matcher-coroutine
  (lambda (tree-cor-1 tree-cor-2)
    (coroutine dont-need-an-init-arg
      (let loop ()
        (let ((leaf1 (resume tree-cor-1 'get-a-life))
               (leaf2 (resume tree-cor-2 'get-a-life)))
          (if (eqv? leaf1 leaf2)
              (if (null? leaf1) #t (loop))
              #f))))))
(define make-leaf-gen-coroutine
  (lambda (tree matcher-cor)
    (coroutine dont-need-an-init-arg
      (let loop ((tree tree))
        (cond ((null? tree) 'skip)
          ((pair? tree)
            (loop (car tree))
            (loop (cdr tree)))
          (else
            (resume matcher-cor tree)))))
    (resume matcher-cor '()))))
(define same-fringe?
  (lambda (tree1 tree2)
    (letrec ((tree-cor-1
                (make-leaf-gen-coroutine
                  tree1
                  (lambda (v) (matcher-cor v))))
             (tree-cor-2
                (make-leaf-gen-coroutine
                  tree2
                  (lambda (v) (matcher-cor v))))
             (matcher-cor
                (make-matcher-coroutine
                  (lambda (v) (tree-cor-1 v))
                  (lambda (v) (tree-cor-2 v))))
             (matcher-cor 'start-ball-rolling))))
> (same-fringe? '(1 2) (3 4) 5) '(((1) 2) 3 (4 5)))
#t
Amb

Amb takes $n$ arguments and returns one of them *ambiguously*. For example, the expression

$$(\mathsf{list} \ (\mathsf{amb} \ 1 \ 2 \ 3) \ (\mathsf{amb} \ 'a \ 'b))$$

can take any of the values

$$(1 \ a) \ (1 \ b) \ (2 \ a) \ (2 \ b) \ (3 \ a) \ (3 \ b)$$
(define amb-fail '*)

(define initialize-amb-fail
  (lambda ()
    (set! amb-fail
      (lambda ()
        (lambda ()
          (error "amb tree exhausted")))))

(initialize-amb-fail)
(define-macro amb
  (lambda alts
    `(let ((+prev-amb-fail amb-fail))
      (call/cc
       (lambda (+sk)
         ,(map (lambda (alt)
            `(call/cc
             (lambda (+fk)
              (set! amb-fail
               (lambda ()
                 (set! amb-fail +prev-amb-fail)
                 (+fk 'fail)))
              (+sk ,alt))))
        alts)
      (+prev-amb-fail))))))
Now we investigate a sample problem for amb... (snarfed from SICP)

We have these five persons Baker, Cooper, Fletcher, Miller, and Smith. They live on five distinct floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher’s. Fletcher does not live on a floor adjacent to Cooper’s. Where does everyone live?
Solution—

(define (distinct? items)
  (cond ((null? items) #t)
        ((null? (cdr items)) #t)
        ((member (car items) (cdr items)) #f)
        (else (distinct? (cdr items))))))
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require
      (distinct? (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
           (list 'cooper cooper)
           (list 'fletcher fletcher)
           (list 'miller miller)
           (list 'smith smith))))
> (multiple-dwelling)
  ((baker 3) (cooper 2) (fletcher 4)
   (miller 5) (smith 1))
References

- Abelson et al. *SICP*

- Dorai Sitaram. *Teach yourself Scheme in Fixnum Days.*
  

- Paul Graham. *ANSI Common Lisp.*

- Friedman et al. *Essentials of Programming Languages.*

- Abelson et al. *R5RS*