call-with-current-continuation

In the Scheme computer programming language, the subroutine or function **call-with-current-continuation**, abbreviated **call/cc**, is used as a control flow operator. It has been adopted by several other programming languages.

Taking a function `f` as its only argument, `(call/cc f)` within an expression is applied to the current continuation of the expression. For example `((call/cc f) e2)` is equivalent to applying `f` to the current continuation of the expression. The current continuation is given by replacing `(call/cc f)` by a variable `c` bound by a lambda abstraction, so the current continuation is `(lambda (c) (c e2))`. Applying the function `f` to it gives the final result `(f (lambda (c) (c e2)))`.

As a complementary example, in an expression `(e1 (call/cc f))`, the continuation for the sub-expression `(call/cc f)` is `(lambda (c) (e1 c))`, so the whole expression is equivalent to `(f (lambda (c) (e1 c)))`. In other words it takes a "snapshot" of the current control context or control state of the program as an object and applies `f` to it. The continuation object is a first-class value and is represented as a function, with function application as its only operation. When a continuation object is applied to an argument, the existing continuation is eliminated and the applied continuation is restored in its place, so that the program flow will continue at the point at which the continuation was captured and *the argument of the continuation* then becomes the "return value" of the call/cc invocation. Continuations created with call/cc may be called more than once, and even from outside the dynamic extent of the call/cc application.

In computer science, making this type of implicit program state visible as an object is termed *reification*. (Scheme does not syntactically distinguish between applying continuations or functions.)

With call/cc a variety of complex control operators can be implemented from other languages via a few lines of code, e.g., McCarthy's `amb` operator for nondeterministic choice, Prolog-style backtracking, Simula 67-style coroutines and generalizations thereof, Icon-style generators, or engines and threads or even the obscure COMEFROM.

Contents

Examples

As shown by the next example, call/cc can be used to emulate the function of the return statement known from C-style languages, which is missing from Scheme:

```
(define (f return)
  (return 2)
  3)

(display (f (lambda (x) x))) ; displays 3

(display (call-with-current-continuation f)) ; displays 2
```

Calling *f* with a regular function argument first applies this function to the value 2, then returns 3. However, when *f* is passed to call/cc (as in the last line of the example), applying the parameter (the continuation) to 2 forces execution of the program to jump to the point where call/cc was called, and causes call/cc to return the value 2. This is then printed by the display function.

In the next example, call/cc is used twice: once to generate a "return" continuation as in the first example and once to suspend an iteration through a list of items:

```
;; [LISTOF X] -> ( -> X u 'you-fell-off-the-end)
(define (generate-one-element-at-a-time lst)
  ;; Both internal functions are closures over lst

  ;; Internal variable/Function which passes the current element in a list
  ;; to its return argument (which is a continuation), or passes an end-of-list marker
  ;; if no more elements are left. On each step the function name is
  ;; rebound to a continuation which points back into the function body,
  ;; while return is rebound to whatever continuation the caller specifies.
  (define (control-state return)
    (for-each
     (lambda (element)
               (set! return (call-with-current-continuation
                             (lambda (resume-here)
                               ;; Grab the current continuation
                              (set! control-state resume-here)
                              (return element))))) ;; (return element) evaluates to next
return
     lst)
    (return 'you-fell-off-the-end))

  ;; (-> X u 'you-fell-off-the-end)
  ;; This is the actual generator, producing one item from a-list at a time.
  (define (generator)
    (call-with-current-continuation control-state))

  ;; Return the generator
  generator)

(define generate-digit
  (generate-one-element-at-a-time '(0 1 2)))

(generate-digit) ;; 0
(generate-digit) ;; 1
(generate-digit) ;; 2
(generate-digit) ;; you-fell-off-the-end
```

Every time the loop is about to process another item from the list, the function grabs the current continuation, and assigns it to the variable 'control-state'. This variable initially is the closure that iterates through all elements of the list. As the computation progresses, it becomes a closure that iterates through a suffix of the given list. While the use of "call/cc" is unnecessary for a linear collection, such as `[LISTOF X]`, the code generalizes to *any* collection that can be traversed.

Call-with-current-continuation can also express other sophisticated primitives. For example, the next sample performs cooperative multitasking using continuations:

```scheme
;; Cooperative multitasking using call-with-current-continuation
;; in 25 lines of scheme

;; The list of threads waiting to run. This is a list of one
;; argument non-returning functions (continuations, mostly)
;; A continuation is a non-returning function, just like (exit),
;; in that it never gives up control to whatever called it.

(define ready-list '())

;; A non-returning function. If there is any other thread
;; waiting to be run, it causes the next thread to run if there
;; is any left to run, otherwise it calls the original exit
;; which exits the whole environment.
(define exit
  ;; The original exit which we override.
  (let ((exit exit))
    ;; The overriding function.
    (lambda ()
      (if (not (null? ready-list))
      ;; There is another thread waiting to be run.
      ;; So we run it.
          (let ((cont (car ready-list)))
            (set! ready-list (cdr ready-list))
        ;; Since the ready-list is only non-returning
        ;; functions, this will not return.
            (cont #f))
      ;; Nothing left to run.
      ;; The original (exit) is a non returning function,
      ;; so this is a non-returning function.
          (exit)))))

;; Takes a one argument function with a given
;; argument and forks it off. The forked function's new
;; thread will exit if/when the function ever exits.
(define (fork fn arg)
  (set! ready-list
        (append ready-list
        ;; This function added to the
        ;; ready-list is non-returning,
        ;; since exit is non returning.
        (list
         (lambda (x)
           (fn arg)
           (exit))))))

;; Gives up control for the next thread waiting to be run.
;; Although it will eventually return, it gives up control
;; and will only regain it when the continuation is called.
(define (yield)
  (call-with-current-continuation
   ;; Capture the continuation representing THIS call to yield
   (lambda (cont)
     ;; Stick it on the ready list
     (set! ready-list
           (append ready-list
                   (list cont)))
     ;; Get the next thread, and start it running.
     (let ((cont (car ready-list)))
       (set! ready-list (cdr ready-list))
       ;; Run it.
       (cont #f)))))
```

In 1999, David Madore (inventor of the Unlambda programming language) accidentally discovered a 12-character Unlambda term, making use of call/cc, that printed all natural numbers sequentially in unary representation: ``` ``r`ci`.*`ci ```.[1] This program and the apparent mystery surrounding its effect have attracted some attention, and are commonly known as the *yin-yang puzzle*.[2] A Scheme translation, provided by Madore, is as follows:

```scheme
(let* ((yin
        ((lambda (cc) (display #\@) cc) (call-with-current-continuation (lambda (c) c))))
```

```
      (yang
        ((lambda (cc) (display #\*) cc) (call-with-current-continuation (lambda (c) c)))))
    (yin yang))
```

Criticism

Oleg Kiselyov, author of a delimited continuation implementation for OCaml, and designer of an application programming interface (API) for delimited stack manipulation to implement control operators, advocates the use of delimited continuations instead of the full-stack continuations that call/cc manipulates: "Offering call/cc as a core control feature in terms of which all other control facilities should be implemented turns out a bad idea. Performance, memory and resource leaks, ease of implementation, ease of use, ease of reasoning all argue against call/cc."[3]

Relation to non-constructive logic

The Curry–Howard correspondence between proofs and programs relates *call/cc* to Peirce's law, which extends intuitionistic logic to non-constructive, classical logic: $((\alpha \to \beta) \to \alpha) \to \alpha$. Here, $((\alpha \to \beta) \to \alpha)$ is the type of the function *f*, which can either return a value of type α directly or apply an argument to the continuation of type $(\alpha \to \beta)$. Since the existing context is deleted when the continuation is applied, the type β is never used and may be taken to be $\bot$, the empty type.

The principle of double negation elimination $((\alpha \to \bot) \to \bot) \to \alpha$ is comparable to a variant of call-cc which expects its argument *f* to always evaluate the current continuation without normally returning a value. Embeddings of classical logic into intuitionistic logic are related to continuation passing style translation.[4]

Languages implementing call/cc

- Scheme
- Racket
- Standard ML[5]
- Haskell in the continuation Monad
- Ruby[6]
- Unlambda
- C++[7]
- R[8]

See also

- Goto
- Continuation-passing style
- Fiber (computer science)

References

1. David Madore, "call/cc mind-boggler" (https://groups.google.com/forum/#!msg/comp.lang.scheme/Fysq_Wplxsw/awxEZ_uxW20J)

2. Yin Wang, "Understanding the Yin-Yang Puzzle" (https://web.archive.org/web/20140129194441/http://yinwang0.wordpress.com/2012/07/27/yin-yang-puzzle)
3. "An argument against call/cc" (http://okmij.org/ftp/continuations/against-callcc.html).
4. Sørensen, Morten Heine; Urzyczyn, Paweł (2007). "Classical Logic and Control Operators". *Lectures on the Curry-Howard isomorphism* (1st ed.). Boston, MA: Elsevier. ISBN 0444520775.
5. "The CONT signature" (http://www.smlnj.org/doc/SMLofNJ/pages/cont.html#SIG:CONT.callcc:VAL). *Standard ML of New Jersey*. Bell Labs, Lucent Technologies. 1997-10-28. Retrieved 2019-05-15.
6. "Class: Continuation" (https://ruby-doc.org/core-2.7.2/Continuation.html). *Ruby-doc.org*. Neurogami, James Britt. Retrieved 2019-05-15.
7. Kowalke, Oliver (2014). "Context switching with call/cc" (https://www.boost.org/doc/libs/1_70_0/libs/context/doc/html/context/cc.html). *Boost.org*. Retrieved 2019-05-15.
8. https://stat.ethz.ch/R-manual/R-devel/library/base/html/callCC.html

External links

- A short introduction to `call-with-current-continuation` (http://community.schemewiki.org/?call-with-current-continuation)
- Definition of `call-with-current-continuation` in the Scheme spec (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_idx_566)
- Humorous explanation of `call-with-current-continuation` from Rob Warnock in Usenet's `comp.lang.lisp` (https://groups.google.com/group/comp.lang.lisp/msg/4e1f782be5ba2841)
- Cooperative multitasking in Scheme using Call-CC (https://web.archive.org/web/20100525110918/http://www.icsi.berkeley.edu/~nweaver/multitask.scm)