20

Continuations

A continuation is a program frozen in action: a single functional object containing the state of a computation. When the object is evaluated, the stored computation is restarted where it left off. In solving certain types of problems it can be a great help to be able to save the state of a program and restart it later. In multiprocessing, for example, a continuation conveniently represents a suspended process. In nondeterministic search programs, a continuation can represent a node in the search tree.

Continuations can be difficult to understand. This chapter approaches the topic in two steps. The first part of the chapter looks at the use of continuations in Scheme, which has built-in support for them. Once the behavior of continuations has been explained, the second part shows how to use macros to build continuations in Common Lisp programs. Chapters 21–24 will all make use of the macros defined here.

20.1 Scheme Continuations

One of the principal ways in which Scheme differs from Common Lisp is its explicit support for continuations. This section shows how continuations work in Scheme. (Figure 20.1 lists some other differences between Scheme and Common Lisp.)

A continuation is a function representing the future of a computation. Whenever an expression is evaluated, something is waiting for the value it will return. For example, in

1. Scheme makes no distinction between what Common Lisp calls the `symbol-value` and `symbol-function` of a symbol. In Scheme, a variable has a single value, which can be either a function or some other sort of object. Thus there is no need for sharp-quote or `funcall` in Scheme. The Common Lisp:

```
(let ((f #'(lambda (x) (1+ x))))
  (funcall f 2))
```

would be in Scheme:

```
(let ((f (lambda (x) (1+ x))))
  (f 2))
```

2. Since Scheme has only one name-space, it doesn't need separate operators (e.g. `defun` and `setq`) for assignments in each. Instead it has `define`, which is roughly equivalent to `defvar`, and `set!`, which takes the place of `setq`. Global variables must be created with `define` before they can be set with `set!`.

3. In Scheme, named functions are usually defined with `define`, which takes the place of `defun` as well as `defvar`. The Common Lisp:

```
(defun foo (x) (1+ x))
```

has two possible Scheme translations:

```
(define foo (lambda (x) (1+ x)))
(define (foo x) (1+ x))
```

4. In Common Lisp, the arguments to a function are evaluated left-to-right. In Scheme, the order of evaluation is deliberately unspecified. (And implementors delight in surprising those who forget this.)

5. Instead of `t` and `nil`, Scheme has `#t` and `#f`. The empty list, `()`, is true in some implementations and false in others.

6. The default clause in `cond` and `case` expressions has the key `else` in Scheme, instead of `t` as in Common Lisp.

7. Several built-in operators have different names: `consp` is `pair?`, `null` is `null?`, `mapcar` is (almost) `map`, and so on. Ordinarily these should be obvious from the context.

Figure 20.1: Some differences between Scheme and Common Lisp.

```
(/ (- x 1) 2)
```

when `(- x 1)` is evaluated, the outer `/` expression is waiting for the value, and something else is waiting for *its* value, and so on and so on, all the way back to the toplevel—where `print` is waiting.

We can think of the continuation at any given time as a function of one argument. If the previous expression were typed into the toplevel, then when the subexpression `(- x 1)` was evaluated, the continuation would be:

```
(lambda (val) (/ val 2))
```

That is, the remainder of the computation could be duplicated by calling this function on the return value. If instead the expression occurred in the following context

```
(define (f1 w)
  (let ((y (f2 w)))
    (if (integer? y) (list 'a y) 'b)))

(define (f2 x)
  (/ (- x 1) 2))
```

and `f1` were called from the toplevel, then when `(- x 1)` was evaluated, the continuation would be equivalent to

```
(lambda (val)
  (let ((y (/ val 2)))
    (if (integer? y) (list 'a y) 'b)))
```

In Scheme, continuations are first-class objects, just like functions. You can ask Scheme for the current continuation, and it will make you a function of one argument representing the future of the computation. You can save this object for as long as you like, and when you call it, it will restart the computation that was taking place when it was created.

Continuations can be understood as a generalization of closures. A closure is a function plus pointers to the lexical variables visible at the time it was created. A continuation is a function plus a pointer to the whole stack pending at the time it was created. When a continuation is evaluated, it returns a value using its own copy of the stack, ignoring the current one. If a continuation is created at T_1 and evaluated at T_2, it will be evaluated with the stack that was pending at T_1.

Scheme programs have access to the current continuation via the built-in operator `call-with-current-continuation` (`call/cc` for short). When a program calls `call/cc` on a function of one argument:

```
(call-with-current-continuation
  (lambda (cc)
    ...))
```

the function will be passed another function representing the current continuation. By storing the value of `cc` somewhere, we save the state of the computation at the point of the `call/cc`.

In this example, we `append` together a list whose last element is the value returned by a `call/cc` expression:

```
> (define frozen)
FROZEN
> (append '(the call/cc returned)
          (list (call-with-current-continuation
                  (lambda (cc)
                    (set! frozen cc)
                    'a))))
(THE CALL/CC RETURNED A)
```

The `call/cc` returns `a`, but first saves the continuation in the global variable `frozen`.

Calling `frozen` will restart the old computation at the point of the `call/cc`. Whatever value we pass to `frozen` will be returned as the value of the `call/cc`:

```
> (frozen 'again)
(THE CALL/CC RETURNED AGAIN)
```

Continuations aren't used up by being evaluated. They can be called repeatedly, just like any other functional object:

```
> (frozen 'thrice)
(THE CALL/CC RETURNED THRICE)
```

When we call a continuation within some other computation, we see more clearly what it means to return back up the old stack:

```
> (+ 1 (frozen 'safely))
(THE CALL/CC RETURNED SAFELY)
```

Here, the pending `+` is ignored when `frozen` is called. The latter returns up the stack that was pending at the time it was first created: through `list`, then `append`, to the toplevel. If `frozen` returned a value like a normal function call, the expression above would have yielded an error when `+` tried to add `1` to a list.

Continuations do not get unique copies of the stack. They may share variables with other continuations, or with the computation currently in progress. In this example, two continuations share the same stack:

```
> (define froz1)
FROZ1
> (define froz2)
FROZ2
> (let ((x 0))
    (call-with-current-continuation
      (lambda (cc)
        (set! froz1 cc)
        (set! froz2 cc)))
    (set! x (1+ x))
    x)
1
```

so calls to either will return successive integers:

```
> (froz2 ())
2
> (froz1 ())
3
```

Since the value of the `call/cc` expression will be discarded, it doesn't matter what argument we give to `froz1` and `froz2`.

Now that we can store the state of a computation, what do we do with it? Chapters 21–24 are devoted to applications which use continuations. Here we will consider a simple example which conveys well the flavor of programming with saved states: we have a set of trees, and we want to generate lists containing one element from each tree, until we get a combination satisfying some condition.

Trees can be represented as nested lists. Page 70 described a way to represent one kind of tree as a list. Here we use another, which allows interior nodes to have (atomic) values, and any number of children. In this representation, an interior node becomes a list; its car contains the value stored at the node, and its cdr contains the representations of the node's children. For example, the two trees shown in Figure 20.2 can be represented:

```
(define t1 '(a (b (d h)) (c e (f i) g)))
(define t2 '(1 (2 (3 6 7) 4 5)))
```

Figure 20.3 contains functions which do depth-first traversals on such trees. In a real program we would want to do something with the nodes as we encountered them. Here we just print them. The function `dft`, given for comparison, does an ordinary depth-first traversal:

```
> (dft t1)
ABDHCEFIG()
```

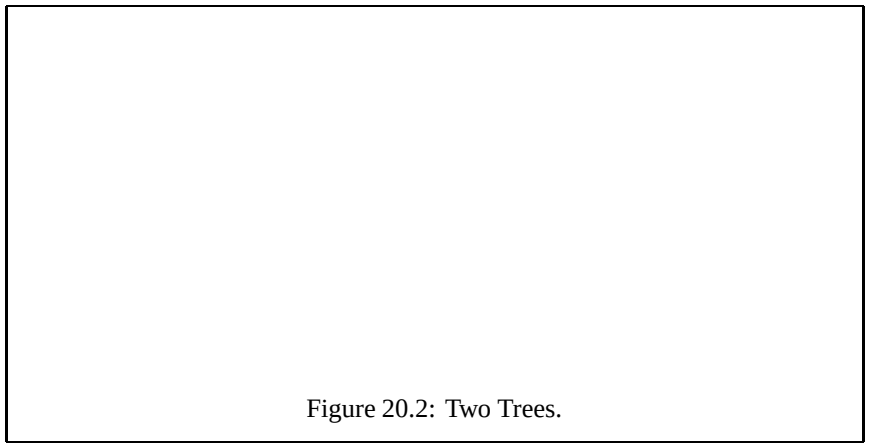

Figure 20.2: Two Trees.

The function `dft-node` follows the same path through the tree, but deals out nodes one at a time. When `dft-node` reaches a node, it follows the car of the node, and pushes onto `*saved*` a continuation to explore the cdr.

```
> (dft-node t1)
A
```

Calling `restart` continues the traversal, by popping the most recently saved continuation and calling it.

```
> (restart)
B
```

Eventually there will be no saved states left, a fact which `restart` signals by returning `done`:

```
⋮
> (restart)
G
> (restart)
DONE
```

Finally, the function `dft2` neatly packages up what we just did by hand:

```
> (dft2 t1)
ABDHCEFIG()
```

```
(define (dft tree)
  (cond ((null? tree) ())
        ((not (pair? tree)) (write tree))
        (else (dft (car tree))
              (dft (cdr tree)))))

(define *saved* ())

(define (dft-node tree)
  (cond ((null? tree) (restart))
        ((not (pair? tree)) tree)
        (else (call-with-current-continuation
                (lambda (cc)
                  (set! *saved*
                        (cons (lambda ()
                                (cc (dft-node (cdr tree))))
                              *saved*))
                  (dft-node (car tree)))))))

(define (restart)
  (if (null? *saved*)
      'done
      (let ((cont (car *saved*)))
        (set! *saved* (cdr *saved*))
        (cont))))

(define (dft2 tree)
  (set! *saved* ())
  (let ((node (dft-node tree)))
    (cond ((eq? node 'done) ())
          (else (write node)
                (restart)))))
```

Figure 20.3: Tree traversal using continuations.

Notice that there is no explicit recursion or iteration in the definition of `dft2`: successive nodes are printed because the continuations invoked by `restart` always return back through the same `cond` clause in `dft-node`.

This kind of program works like a mine. It digs the initial shaft by calling `dft-node`. So long as the value returned is not `done`, the code following the call

to `dft-node` will call `restart`, which sends control back down the stack again. This process continues until the return value signals that the mine is empty. Instead of printing this value, `dft2` returns `#f`. Search with continuations represents a novel way of thinking about programs: put the right code in the stack, and get the result by repeatedly returning up through it.

If we only want to traverse one tree at a time, as in `dft2`, then there is no reason to bother using this technique. The advantage of `dft-node` is that we can have several instances of it going at once. Suppose we have *two* trees, and we want to generate, in depth-first order, the cross-product of their elements.

```
> (set! *saved* ())
()
> (let ((node1 (dft-node t1)))
    (if (eq? node1 'done)
        'done
        (list node1 (dft-node t2))))
(A 1)
> (restart)
(A 2)
⋮
> (restart)
(B 1)
⋮
```

Using normal techniques, we would have had to take explicit steps to save our place in the two trees. With continuations, the state of the two ongoing traversals is maintained automatically. In a simple case like this one, saving our place in the tree would not be so difficult. The trees are permanent data structures, so at least we have some way of getting hold of "our place" in the tree. The great thing about continuations is that they can just as easily save our place in the middle of *any* computation, even if there are no permanent data structures associated with it. The computation need not even have a finite number of states, so long as we only want to restart a finite number of them.

As Chapter 24 will show, both of these considerations turn out to be important in the implementation of Prolog. In Prolog programs, the "search trees" are not real data structures, but are implicit in the way the program generates results. And the trees are often infinite, in which case we cannot hope to search the whole of one before searching the next; we have no choice but to save our place, one way or another.

20.2 Continuation-Passing Macros

Common Lisp doesn't provide `call/cc`, but with a little extra effort we can do the same things as we can in Scheme. This section shows how to use macros to build continuations in Common Lisp programs. Scheme continuations gave us two things:

1. The bindings of all variables at the time the continuation was made.
2. The state of the computation—what was going to happen from then on.

In a lexically scoped Lisp, closures give us the first of these. It turns out that we can also use closures to maintain the second, by storing the state of the computation in variable bindings as well.

The macros shown in Figure 20.4 make it possible to do function calls while preserving continuations. These macros replace the built-in Common Lisp forms for defining functions, calling them, and returning values.

○ Functions which want to use continuations (or call functions which do) should be defined with `=defun` instead of `defun`. The syntax of `=defun` is the same as that of `defun`, but its effect is subtly different. Instead of defining just a function, `=defun` defines a function and a macro which expands into a call to it. (The macro must be defined first, in case the function calls itself.) The function will have the body that was passed to `=defun`, but will have an additional parameter, `*cont*`, consed onto its parameter list. In the expansion of the macro, this function will receive `*cont*` along with its other arguments. So

```
(=defun add1 (x) (=values (1+ x)))
```

macroexpands into

```
(progn (defmacro add1 (x)
         `(=add1 *cont* ,x))
       (defun =add1 (*cont* x)
         (=values (1+ x))))
```

When we call `add1`, we are actually calling not a function but a macro. The macro expands into a function call,[1] but with one extra parameter: `*cont*`. So the current value of `*cont*` is always passed implicitly in a call to an operator defined with `=defun`.

What is `*cont*` for? It will be bound to the current continuation. The definition of `=values` shows how this continuation will be used. Any function defined using `=defun` must return with `=values`, or call some other function

[1] Functions created by `=defun` are deliberately given interned names, to make it possible to `trace` them. If tracing were never necessary, it would be safer to gensym the names.

```
(setq *cont* #'identity)

(defmacro =lambda (parms &body body)
  `#'(lambda (*cont* ,@parms) ,@body))

(defmacro =defun (name parms &body body)
  (let ((f (intern (concatenate 'string
                                "=" (symbol-name name)))))
    `(progn
       (defmacro ,name ,parms
         `(,',f *cont* ,,@parms))
       (defun ,f (*cont* ,@parms) ,@body))))

(defmacro =bind (parms expr &body body)
  `(let ((*cont* #'(lambda ,parms ,@body))) ,expr))

(defmacro =values (&rest retvals)
  `(funcall *cont* ,@retvals))

(defmacro =funcall (fn &rest args)
  `(funcall ,fn *cont* ,@args))

(defmacro =apply (fn &rest args)
  `(apply ,fn *cont* ,@args))
```

Figure 20.4: Continuation-passing macros.

which does so. The syntax of `=values` is the same as that of the Common Lisp form `values`. It can return multiple values if there is an `=bind` with the same number of arguments waiting for them, but can't return multiple values to the toplevel. ○

The parameter `*cont*` tells a function defined with `=defun` what to do with its return value. When `=values` is macroexpanded it will capture `*cont*`, and use it to simulate returning from the function. The expression

```
> (=values (1+ n))
```

expands into

```
(funcall *cont* (1+ n))
```

At the toplevel, the value of *cont* is identity, which just returns whatever is passed to it. When we call (add1 2) from the toplevel, the call gets macroexpanded into the equivalent of

```
(funcall #'(lambda (*cont* n) (=values (1+ n))) *cont* 2)
```

The reference to *cont* will in this case get the global binding. The =values expression will thus macroexpand into the equivalent of:

```
(funcall #'identity (1+ n))
```

which just adds 1 to n and returns the result.

In functions like add1, we go through all this trouble just to simulate what Lisp function call and return do anyway:

```
> (=defun bar (x)
    (=values (list 'a (add1 x))))
BAR
> (bar 5)
(A 6)
```

The point is, we have now brought function call and return under our own control, and can do other things if we wish.

It is by manipulating *cont* that we will get the effect of continuations. Although *cont* has a global value, this will rarely be the one used: *cont* will nearly always be a parameter, captured by =values and the macros defined by =defun. Within the body of add1, for example, *cont* is a parameter and not the global variable. This distinction is important because these macros wouldn't work if *cont* were not a local variable. That's why *cont* is given its initial value in a setq instead of a defvar: the latter would also proclaim it to be special.

The third macro in Figure 20.4, =bind, is intended to be used in the same way as multiple-value-bind. It takes a list of parameters, an expression, and a body of code: the parameters are bound to the values returned by the expression, and the code body is evaluated with those bindings. This macro should be used whenever additional expressions have to be evaluated after calling a function defined with =defun.

```
> (=defun message ()
    (=values 'hello 'there))
MESSAGE
```

```
> (=defun baz ()
    (=bind (m n) (message)
      (=values (list m n))))
BAZ
> (baz)
(HELLO THERE)
```

Notice that the expansion of an `=bind` creates a new variable called `*cont*`. The body of `baz` macroexpands into:

```
(let ((*cont* #'(lambda (m n)
                  (=values (list m n)))))
  (message))
```

which in turn becomes:

```
(let ((*cont* #'(lambda (m n)
                  (funcall *cont* (list m n)))))
  (=message *cont*))
```

The new value of `*cont*` is the body of the `=bind` expression, so when `message` "returns" by funcalling `*cont*`, the result will be to evaluate the body of code. However (and this is the key point), within the body of the `=bind`:

```
#'(lambda (m n)
    (funcall *cont* (list m n)))
```

the `*cont*` that was passed as an argument to `=baz` is still visible, so when the body of code in turn evaluates an `=values`, *it* will be able to return to the original calling function. The closures are knitted together: each binding of `*cont*` is a closure containing the previous binding of `*cont*`, forming a chain which leads all the way back up to the global value.

We can see the same phenomenon on a smaller scale here:

```
> (let ((f #'identity))
    (let ((g #'(lambda (x) (funcall f (list 'a x)))))
      #'(lambda (x) (funcall g (list 'b x)))))
#<Interpreted-Function BF6326>
> (funcall * 2)
(A (B 2))
```

This example creates a function which is a closure containing a reference to `g`, which is itself a closure containing a reference to `f`. Similar chains of closures were built by the network compiler on page 80.

1. The parameter list of a function defined with `=defun` must consist solely of parameter names.

2. Functions which make use of continuations, or call other functions which do, must be defined with `=lambda` or `=defun`.

3. Such functions must terminate either by returning values with `=values`, or by calling another function which obeys this restriction.

4. If an `=bind`, `=values`, `=apply`, or `=funcall` expression occurs in a segment of code, it must be a tail call. Any code to be evaluated after an `=bind` should be put in its body. So if we want to have several `=bind`s one after another, they must be nested:

```
(=defun foo (x)
  (=bind (y) (bar x)
    (format t "Ho ")
    (=bind (z) (baz x)
      (format t "Hum.")
      (=values x y z))))
```

Figure 20.5: Restrictions on continuation-passing macros.

The remaining macros, `=apply` and `=funcall`, are for use with functions defined by `=lambda`. Note that "functions" defined with `=defun`, because they are actually macros, cannot be given as arguments to `apply` or `funcall`. The way around this problem is analogous to the trick mentioned on page 110. It is to package up the call inside another `=lambda`:

```
> (=defun add1 (x)
    (=values (1+ x)))
ADD1
> (let ((fn (=lambda (n) (add1 n))))
    (=bind (y) (=funcall fn 9)
      (format nil "9 + 1 = ~A" y)))
"9 + 1 = 10"
```

Figure 20.5 summarizes all the restrictions imposed by the continuation-passing macros. Functions which neither save continuations, nor call other functions which do, need not use these special macros. Built-in functions like `list`, for example, are exempt.

Figure 20.6 contains the code from Figure 20.3, translated from Scheme into Common Lisp, and using the continuation-passing macros instead of Scheme

```
(defun dft (tree)
  (cond ((null tree) nil)
        ((atom tree) (princ tree))
        (t (dft (car tree))
           (dft (cdr tree)))))

(setq *saved* nil)

(=defun dft-node (tree)
  (cond ((null tree) (restart))
        ((atom tree) (=values tree))
        (t (push #'(lambda () (dft-node (cdr tree)))
                 *saved*)
           (dft-node (car tree)))))

(=defun restart ()
  (if *saved*
      (funcall (pop *saved*))
      (=values 'done)))

(=defun dft2 (tree)
  (setq *saved* nil)
  (=bind (node) (dft-node tree)
    (cond ((eq node 'done) (=values nil))
          (t (princ node)
             (restart)))))
```

Figure 20.6: Tree traversal using continuation-passing macros.

continuations. With the same example tree, `dft2` works just as before:

```
> (setq t1 '(a (b (d h)) (c e (f i) g))
        t2 '(1 (2 (3 6 7) 4 5)))
(1 (2 (3 6 7) 4 5))
> (dft2 t1)
ABDHCEFIG
NIL
```

Saving states of multiple traversals also works as in Scheme, though the example becomes a bit longer:

```
> (=bind (node1) (dft-node t1)
    (if (eq node1 'done)
        'done
        (=bind (node2) (dft-node t2)
          (list node1 node2))))
(A 1)
> (restart)
(A 2)
⋮
> (restart)
(B 1)
⋮
```

By knitting together a chain of lexical closures, Common Lisp programs can build their own continuations. Fortunately, the closures are knitted together by the macros in the sweatshop of Figure 20.4, and the user can have the finished garment without giving a thought to its origins.

Chapters 21–24 all rely on continuations in some way. These chapters will show that continuations are an abstraction of unusual power. They may not be overly fast, especially when implemented on top of the language as macros, but the abstractions we can build upon them make certain programs much faster to write, and there is a place for that kind of speed too.

20.3 Code-Walkers and CPS Conversion

The macros described in the previous section represent a compromise. They give us the power of continuations, but only if we write our programs in a certain way. Rule 4 in Figure 20.5 means that we always have to write

```
(=bind (x) (fn y)
  (list 'a x))
```

rather than

```
(list 'a                                              ; wrong
      (=bind (x) (fn y) x))
```

A true `call/cc` imposes no such restrictions on the programmer. A `call/cc` can grab the continuation at any point in a program of any shape. We could implement an operator with the full power of `call/cc`, but it would be a lot more work. This section outlines how it could be done.

A Lisp program can be transformed into a form called "continuation-passing style." Programs which have undergone complete CPS conversion are impossible to read, but one can grasp the spirit of this process by looking at code which has been partially transformed. The following function to reverse lists: ○

```
(defun rev (x)
  (if (null x)
      nil
      (append (rev (cdr x)) (list (car x)))))
```

yields an equivalent continuation-passing version:

```
(defun rev2 (x)
  (revc x #'identity))

(defun revc (x k)
  (if (null x)
      (funcall k nil)
      (revc (cdr x)
            #'(lambda (w)
                (funcall k (append w (list (car x))))))))
```

In the continuation-passing style, functions get an additional parameter (here `k`) whose value will be the continuation. The continuation is a closure representing what should be done with the current value of the function. On the first recursion, the continuation is `identity`; what should be done is that the function should just return its current value. On the second recursion, the continuation will be equivalent to:

```
#'(lambda (w)
    (identity (append w (list (car x)))))
```

which says that what should be done is to append the car of the list to the current value, and return it.

Once you can do CPS conversion, it is easy to write `call/cc`. In a program which has undergone CPS conversion, the entire current continuation is always present, and `call/cc` can be implemented as a simple macro which calls some function with it as an argument.

To do CPS conversion we need a *code-walker*, a program that traverses the trees representing the source code of a program. Writing a code-walker for Common Lisp is a serious undertaking. To be useful, a code-walker has to do ○ more than simply traverse expressions. It also has to know a fair amount about what the expressions mean. A code-walker can't just think in terms of symbols,

for example. A symbol could represent, among other things, itself, a function, a variable, a block name, or a tag for `go`. The code-walker has to use the context to distinguish one kind of symbol from another, and act accordingly.

Since writing a code-walker would be beyond the scope of this book, the macros described in this chapter are the most practical alternative. The macros in this chapter split the work of building continuations with the user. If the user writes programs in something sufficiently close to CPS, the macros can do the rest. That's what rule 4 really amounts to: if everything following an `=bind` expression is within its body, then between the value of `*cont*` and the code in the body of the `=bind`, the program has enough information to construct the current continuation.

The `=bind` macro is deliberately written to make this style of programming feel natural. In practice the restrictions imposed by the continuation-passing macros are bearable.