Lecture 21: Programming with Continuations

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 call/cc: First-Class Continuations

- So far, continuations have existed in the background of our languages, helping us implement control-flow structure like exceptions and non-determinism.
- In this lecture we will turn to how continuations can be used in day-to-day programming.
- However, some languages support **first-class continuations**: i.e., it is possible for a program to programs to acquire its own continuation as a function that can be called.[1]
- Racket is a language that supports this feature: it has a built-in function `call/cc` (short for *call with current continuation*) that takes a function as an argument and calls it with the current continuation for the *entire Racket program*. Calling it looks like this:

```
> (call/cc (lambda (k) (k 1)))
1
```

- The argument `k` in the above lambda is the current continuation. The way to think about it is: whatever `k` is called with is what the entire expression `(call/cc (lambda (k) (k 1)))` evaluates to. That's why the above example evaluated to 1: this is what we called the current continuation with.
- Similar to our very first example of using continuations in our interpreters, we can use `call/cc` to implement a Python-style `return` construct:

```
> (define (f return)
  (let [(_ (return 2))]
    3))

> (display (f (lambda (x) x)))
3
> (display (call/cc f))
2
```

- Notice what's happening in these two cases. In the first case, `return` is the identity function, so its return value is ignored by the `begin` inside `f`. In the second case, `return` is the current continuation, so when it is called control immediately jumps to the `call/cc` and evaluates to 2.

[1] First-class continuations is a fairly rare feature, and not very many programming languages support it (mostly due to the performance overhead of implementing it). However, it is always possible to add first-class continuations into your program by apply continuation-passing transformations, similar to how we compiled away exceptions. There are many varieties of continuations besides `call/cc`, some of which offer different tradoeffs between performance and expressivity. For example, `shift/reset` gives a form of *delimited continuation* that applies to only part of the program; it's less expressive than `call/cc`, but much easier to implement efficiently.

2 Evaluating `call/cc`

- Let's walk through an example of slowly evaluating `call/cc` by hand:

```
(define (f return)
  (let [(_ (return 2))]
    3))

(display (call/cc f))
-->
(f (λ (r) (display r)))
-- substitute in argument -->
(let [(_ ((λ (r) (display r)) 2))]
  3)
-->
2
```

- Notice how, in the above, the invocation of `call/cc` was replaced by the result of the continuation.

- **Note** that the continuation created by `call/cc` is a special kind of function that behaves differently from all other Racket functions: *it never returns control to its caller*.

3 Implementing Exceptions with `call/cc`

- `call/cc` lets you add very exotic control-flow structures into Racket. For instance, we can implement named exception handlers in Racket using `call/cc` without relying on Racket's built in exceptions:

```
;;; runs the `try` function and jumps to `handler` in the
;;; event an exception is raised
(define (run-with-handler try handler)
  (call/cc
   (λ (currcc)
     (try (λ () (currcc handler))))))

; runs to 25
(run-with-handler
 (λ (raiseexn)
   (+ 10 (raiseexn)))
 25)

; runs to 50
(run-with-handler
 (λ (raiseexn)
   (run-with-handler (λ (raiseexn2)
                       (+ 30 (raiseexn2)))
                     50))
 30)
```

- Not all languages support `call/cc` as a built-in construct, but it can be added to any language by writing your programs in continuation-passing style. In fact, to implement `call/cc`, Racket will automatically translate your program into continuation-passing style.

4 Implementing Generators with call/cc

- call/cc is a very powerful control-flow primitive that enables implementing some surprising features.[2]

- A generator is a commonly used design pattern for iterating over datastructures. Here is an example of using generators in Python to iterate over a list of numbers. Generators are defined using a "yield" primitive, which gives up execution to the caller:

```
def generate_numbers(n):
    for i in range(n):
        yield i        # yield is a built-in keyword in Python

# Create a generator object
numbers = generate_numbers(5)

# Iterate over the generator
for num in numbers:
    print(num)
```

- Not all languages support generators in this way. Using call/cc, we can add generators to Racket!

[2]This section is inspired by this blog post from Matt Might: https://matt.might.net/articles/programming-with-continuations--exceptions-backtracking-search-threads-generators-coroutines/

```
#lang racket
(require rackunit)

(define (make-generator generator)
  ; generator process
  (define gen-kont (box '()))

  ; client process
  (define client-kont (box '()))

  ; resumes the generator process
  (define (f new-client-k)
    (set-box! client-kont new-client-k)
    (match (unbox gen-kont)
      ['()
       ; start the generator process
       (generator
        ; this function is called whenever a user calls yield
        (λ (v) (call/cc (λ (new-gen-k)
                          ; store in j the current generator state
                          (set-box! gen-kont new-gen-k)
                          ; resumes the client and passes it the yielded value
                          ((unbox client-kont) v)))))]
      [k (k)]))

  ; return a thunk that steps the generator process
  (λ ()
    (call/cc f)))

(define test
  (make-generator
   (lambda (yield)
     (yield 1)
     (yield 5)
     (yield 10)
     #f)))

(check-equal? (list (test) (test) (test)) '(1 5 10))
```

5 Effects and Effect Handlers

- One of the issues with `call/cc` is that it is very expensive to implement in practice.
- **Effect handlers** give a very powerful and flexible way of interacting with continuations that is *delimited* in scope, meaning that the continuation doesn't simply grab the entire call-stack (as it does with `call/cc`)
- Effect handlers look and feel a lot like exceptions, except they can be resumed by calling their continuation.
- The **effect** keyword declares a new kind of effect. In this case, it declares a `yield` effect that has a single argument `val`.

```
#lang racket
(require rackunit)
(require effect-racket)

(struct box (default))

(effect return (v))

(define (return-service)
  (handler
   [(return v)
    v]))

(with ([return-service])
      (return "oops")
      (display "hello world"))

; returns v if it's less than 5, otherwise it keeps running your program :)
(define (return-if-service)
  (handler
   [(return v)
    (if (> 5 v)
        v
        ; calling `continue' resumes execution from where the effect occurred
        (continue v))]))

(with ([return-if-service])
      (displayln (return 10))
      (return 4)
      (displayln "hello world"))
```

6 Amb with Effect Handlers

- One of the very powerful thing about effect handler continuations is that they can be invoked more than once. This makes it possible to implement `amb` for *all of Racket* quite easily!

```
(effect choice ())
(effect fail ())

(define (amb-service)
  (handler
   [(choice)
    (append (continue #t) (continue #f))
    ]
   [(fail) '()]))

(with [(amb-service)]
      (define a (choice))
      (define b (choice))
      (if (and a b) (fail) (list (cons a b))))
```

7 Implementing Generators with Effect Handlers

- Here is an example of implementing our generator using the Racket effects library:

```
#lang racket

(require effect-racket)

(define (make-generator proc)
  (effect yield (val))
  (define kont #f)
  (define yield-handler
    (handler
     [(yield val)
      (set! kont continue)
      val]))
  (λ ()
    (with (yield-handler)
      (if kont (kont (void)) (proc yield)))))

(define test
  (make-generator
   (lambda (yield)
     (yield 1)
     (yield 5)
     (yield 10)
     #f)))
```

8 Continuations Conclusion