Racket: Continuations

Advanced Functional Programming

Jean-Noël Monette

November 2013

Today

- Finishing previous lecture
- Continuations
  - Continuation Passing Style
  - First Class Continuations.
  - Examples

Finishing previous lecture

- Numbers
- `values`
- Comprehensions

Plus forgotten stuffs

- Infix Notation
- Printing

Dots and Infix Notation

Remember a "fake" list is displayed like that:

```
'(1 2 . 3)
```

One can also use it when entering a list:

`'(1 2 . (3 4))` is equivalent to the list `'(1 2 3 4)`

One can also use two dots around an element of the s-expr to make it the first one.

```
(code (4 . + . 5)) " is transformed into " (code (+ 4 5))
```

This can be useful if you are not comfortable with the prefix notation.

Printing data

There are (at least) three ways to output data (to the console).

- `display` removes all quotation marks and string delimiters.
- `print` does not remove any quotation marks or string delimiters.
- `write` removes the outermost quotation mark if any.

In addition, `(newline)` prints a newline.

```
(displayln '(a "azer" 3))
(print '(a "azer" 3))
(newline)
(write '(a "azer" 3))
```

Parameters

Parameters are variables that can be dynamically bound.

```
(define color (make-parameter "Blue"))
(define (best-color) (display (color)))
(best-color)
(parameterize ([color "red"])
  (best-color))
(best-color)
```

This is preferable to `set!` for several reasons (tail calls, threads, exceptions).

There exist parameters for instance to define the output stream, the level of details when reporting an error...

What is a Continuation?

At any point during execution, it is the part of the computation that remains to do.

In `(+ 1 (+ (+ 3 4) 2))`, when `(+ 3 4)` has been evaluated, the continuation is `(+ 1 (+ [] 2))`, where `[]` represents what has already been done.

How is it any useful?

Two main ways to use the concept:

- Use Continuation Passing Style (CPS) to make continuation explicit.
- Make continuations first-class, so that they can be manipulated.

Use Cases

With continuations, one can implement...

- Exceptions
- Non-deterministic choice points.
- (Lazy) generators
- Coroutines
- ...

Continuation-Passing Style

We want to make the continuations manipulable.

The idea is that to each procedure we pass its continuation.

That is, each procedure has an additional argument that is a procedure of one argument that will be applied to its result.

No procedure should ever return, as returning a value is replaced by calling the continuation.

To work, this relies on the fact that tail calls are optimized away.

An example

```
  (define (sum lst)
    (cond [(cons? lst)
           (+ (car lst) (sum (cdr lst)))]
          [else 0]))
  (display (sum '(1 3 5 -2 17)))

(define (sum lst cont)
  (cond [(cons? lst)
         (sum (cdr lst)
              (lambda (acc)
                (cont (+ (car lst) acc))))]
        [else (cont 0)]))
(sum '(1 3 5 -2 17) display)
```

An example

```
(define (sum-help lst acc)
  (cond [(cons? lst)
         (sum-help (cdr lst) (+ (car lst) acc))]
        [else acc]))
(define (sum lst) (sum-help lst 0))
(display (sum '(1 3 5 -2 17)))
```

```
(define (sum-help lst acc cont)
  (cond [(cons? lst)
         (sum-help (cdr lst)
                   (+ (car lst) acc) cont)]
        [else (cont acc)]))
(define (sum lst cont) (sum-help lst 0 cont))
(sum '(1 3 5 -2 17) display)
```

Is that all?

Almost...

There is a bunch of things missing here: `cond`, `+`, `car` and so on ... should also be in CPS.

```
(define (+& x y cont) (cont (+ x y)))
(define (cdr& lst cont) (cont (cdr lst)))
(define (if& test iftrue iffalse)
  (if test (iftrue) (iffalse)))
(define (=& x y cont) (cont (= x y)))
```

In CPS, exceptions are explicit.

```
(define (/& x y cont error)
  (=& 0 y (lambda (e)
            (if& e
                 error
                 (lambda () (cont (/ x y)))))))

(/& 3 0
    display
    (lambda () (display "div by zero")))
```

Isn’t CPS very convoluted?

- CPS is barely used "pure".
- It is easy to compile non-CPS code into CPS code.
- CPS code is a good medium to apply compiler optimization.
- CPS can be used when returning more than one value.
- CPS is a way to emulate first-class continuations when they do not exist.
- The style can also be used for asynchronous communication.

CPS as callbacks.

A callback function is a procedure to be executed on request.

- In a GUI, the procedure could be called when a button is clicked.
- In a distributed system, the procedure could be called when a mesage is received.
- In a multithreaded system, the procedure could be called when another thread is done.
- ...
- One can see a continuation as a special case of a callback.

First Class Continuations

Imagine you can save the current continuation. What can you do with it?

- Restore it later to escape the current computation.
- A (first-class) continuation is represented as a procedure of one argument.
- Avoid the cluttering of CPS.

call/cc

The (main) way to capture a continuation in Racket is `call-with-current-continuation` or `call/cc` in short.

`call/cc` is a procedure of one argument that must be a procedure (call it "p") of one argument. This argument is the continuation (which is a procedure of one argument, right?).

"p" is run directly and its return value is returned by `call/cc`

Time for some examples.

Examples

`(call/cc (lambda (_) (+ 3 4)))` simply returns 7.

`(call/cc (lambda (cc) cc))` returns the continuation.

`(+ 2 (call/cc (lambda (cc) (+ 30 (cc 10)))))` returns 12.

`(call/cc (lambda (cc) (cc cc)))` returns the continuation as well.

Saving a continuation (1)

```
(define sc #f)
(define (save! x)
  (call/cc (lambda (cc) (set! sc cc) x)))
```

`save!` saves the current continuation into `sc`, which can be used to replay the computation with another value.

```
(+ 1 (+ (+ (save! 3) 4) 2))
(sc 10)
(sc 20)
```

`sc` can be seen as the following procedure:

```
(lambda (XXX) (+ 1 (+ (+ XXX 4) 2)))
```

Saving a continuation. (2)

```
(define (current-continuation)
  (call/cc (lambda (cc) cc)))
```

This procedure returns the current continuation.

What do the following programs do?

```
(let ([cc (current-continuation)])
  (cc cc))

(let ([cc (current-continuation)])
  (cc 0))
```

Continuations as GOTO

```
(define (label)
  (call/cc (lambda (cc) cc)))
(define (goto label)
  (label label))

 (define c 0)
 (let ([l (label)])
   (displayln c)
   (set! c (add1 c))
   (when (< c 10) (goto l)))
```

Continuations as GOTO: Explanations

```
(define (save-where-I-am)
  (call/cc (lambda (cc) cc)))
(define (goto somewhere)
  (somewhere somewhere))

(define c 0)
(let ([l (save-where-I-am)])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l)))
```

`save-where-I-am` saves the current continuation, that is the point where we are in the program.

`goto` jumps directly to the saved point by calling the continuation.

Continuations as GOTO: Explanations (2)

```
(define (save-where-I-am)
  (call/cc (lambda (cc) cc)))
(define (goto somewhere)
  (somewhere somewhere))

(define c 0)
(let ([l (save-where-I-am)])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l)))
```

Let's run the program...

Continuations as GOTO: Explanations (2)

```
(define (save-where-I-am)
  (call/cc (lambda (cc) cc)))
(define (goto somewhere)
  (somewhere somewhere))

(define c 0)
(let ([l (save-where-I-am)])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l)))
```

First `c` is defined to be `0`.

Then `save-where-I-am` is called. Its result is put in `l`.

```
l: (lambda (XXX)
     (let ([l1 XXX])
       (displayln c)
       (set! c (add1 c))
       (when (< c 10) (goto l1))))
```

The body of `l` is the rest of the code, but `(save-where-I-am)` has been replaced by the value passed to the continuation (named `XXX`).

Continuations as GOTO: Explanations (2)

```
(define (save-where-I-am)
  (call/cc (lambda (cc) cc)))
(define (goto somewhere)
  (somewhere somewhere))

(define c 0)
(let ([l (save-where-I-am)])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l)))
```

The value of `c` (i.e. `0`) is printed.

The value of `c` is incremented.

`(goto l)` is called.

`(l l)` is thus called.

The body of `l` is (with `XXX` substituted):

```
(let ([l1 l])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l1)))
```

Continuations as GOTO: Explanations (2)

```
(define (save-where-I-am)
  (call/cc (lambda (cc) cc)))
(define (goto somewhere)
  (somewhere somewhere))

(define c 0)
(let ([l (save-where-I-am)])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l)))
```

The body of `l` is (with `XXX` substituted):

```
(let ([l1 l])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l1)))
```

So `l1` is bound to the same procedure:

```
l1: (lambda (XXX)
      (let ([l2 XXX])
        (displayln c)
        (set! c (add1 c))
        (when (< c 10) (goto l2))))
```

I changed the name of `l` to `l1` and `l2` to insist on lexical scoping. `c` is not renamed as it comes from an outer scope.

Continuations as GOTO: Explanations (2)

```
(define (save-where-I-am)
  (call/cc (lambda (cc) cc)))
(define (goto somewhere)
  (somewhere somewhere))

(define c 0)
(let ([l (save-where-I-am)])
  (displayln c)
  (set! c (add1 c))
  (when (< c 10) (goto l)))
```

If we go on, the body of `l`, then `l1`, `l2`, and so on is executed.

At some point, in `l9`, the guard `(< c 10)` becomes false, and the procedure simply finishes.

Continuations as GOTO

Everyone knows that GOTO statements are really bad...

So let’s look at more meaningful examples.

Exceptions

Exceptions are just continuations in disguise: when there is a problem, we just escape from the current computation.

```
(define exception (make-parameter raise))
(define (try thunk recovery)
  (parameterize ([exception (current-continuation)])
    (cond [(continuation? (exception)) (thunk)]
          [else (recovery (exception))])))
(define (error msg)
  ((exception) msg))
```

Exceptions

```
(try
 (lambda ()
   (displayln 123)
   (error 'oups)
   (displayln 456))
 (lambda (x)
   (display "caught: ")
   (displayln x)))
(error 'Szut)
```

Limitations: no "finally" clause, heavy syntax.

Exceptions are really implemented with continuations under the hood.

Example: Non-deterministic choice

A non-deterministic choice is a point in the program where several alternatives can be chosen. The actual choice is only made at runtime.

We will use this with a `fail` procedure that can be called to tell that an alternative is not viable.

This is very useful to describe backtracking algorithms in simple terms.

```
(define x (choose 'a 'b))
(when (= x 'a) (fail))
(display x)
```

Non-deterministic choice: first version

```
(define (choose a b)
  (let ([cc (current-continuation)])
    (cond [(continuation? cc) (values a cc)]
          [else (values b #f)])))

(define (fail cc)
  (when (continuation? cc) (cc #f)))

(define (run)
  (define-values (val cc) (choose 3 4))
  (displayln val)
  (fail cc))
```

Notes

- First class continuations are not easy to grasp.
- To understand the previous examples (exceptions, non-determinism), walk through them as we did for the goto example.
- The slides after this one have not been covered in the lecture but are there for you to play with if you wish.

Non-deterministic choice: more than 2 alternatives

```
(define (choose a . rest)
  (let ([cc (current-continuation)])
    (cond [(null? rest) (values a #f)]
          [(continuation? cc) (values a cc)]
          [else (apply choose rest)])))

(define (fail cc)
  (when (continuation? cc) (cc #f)))

(define (run)
  (define-values (val cc) (choose 3 4 5 6))
  (displayln val)
  (fail cc))
```

Non-deterministic choice: imbricated choices

```
(define (choose stack a . rest)
  (let ([cc (current-continuation)])
    (cond [(null? rest) (values a stack)]
          [(continuation? cc)
           (values a (cons cc stack))]
          [else (apply choose cc rest)])))

(define (fail stack)
  (cond [(null? stack) (raise 'empty-stack)]
        [else (let ([cc (car stack)])
                (cc (cdr stack)))]))

(define (run)
  (let*-values ([(stack) '()]
                [(val stack) (choose stack 3 4 5 6)]
                [(val2 stack) (choose stack 'a 'b 'c 'd)])
    (displayln (cons val val2))
    (fail stack)))
```

Getting rid of the continuation argument

```
(define stack '())
(define (choose a . rest)
  (let ([cc (current-continuation)])
    (cond [(null? rest) a]
          [(continuation? cc)
           (set! stack (cons cc stack))
           a]
          [else (apply choose rest)])))

(define (fail)
  (cond [(null? stack) (raise 'empty-stack)]
        [else (let ([cc (car stack)])
                (set! stack (cdr stack))
                (cc #f))]))
```

Non-deterministic choice: Examples

```
(define a (choose 3 4 5 6))
(define b (choose 7 6 5 4))
(displayln `(,a ,b))
(fail)

(define a (choose 3 4 5 6))
(define b (choose 7 6 5 4))
(when (< 20 (* a b)) (fail))
(when (> 15 (* a b)) (fail))
(display (cons a b))
```

First-Class Continuations

To wrap up...

- A very powerful mechanism.
- Can seem a bit magical at time.
- Not seen here:
  - Implementing generators.
  - Implementing coroutines.
  - Implementing context switch.
  - Prompts and Delimited Continuations
  - Composable Continuations
  - Escape Continuations

Sources

Ideas of examples coming from the following article

- http://matt.might.net/articles/programming-with-continuations--exceptions-backtracking-search-threads-generators-coroutines/