CS450

Structure of Higher Level Languages

Lecture 19: Continuations

Tiago Cogumbreiro

Today we will...

- Revisit dynamic binding
- Introduce continuations

> Inspired by Professor Konstantinos Sagonas' 2013 lecture on continuations, Uppsala University, Sweeden.

Static versus dynamic scoping

Static Scoping

> **Static binding:** variables are captured at creation time

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))

(check-equal? (g) (+ 23 1))
```

Dynamic Scoping

> **Dynamic binding:** variables depends on the calling context

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))
; NOT VALID RACKET CODE
(check-equal? (g) (+ 23 20))
```

Why dynamic scoping?

1. A controlled way to represent global variables
2. A technique to make code testable

Dynamic scoping example

Dynamic scoping In Racket

```
(define x (make-parameter 1))
(define (f y) (+ y (x)))

(define (g)
  (parameterize ([x 20])
    (define y 3)
    (f (+ (x) y))))

(check-equal? (g) (+ 23 20))
```

Pseudo-Racket dynamic scoping

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))
; NOT VALID RACKET CODE
(check-equal? (g) (+ 23 20))
```

- Function `make-parameter` returns a reference to a dynamically scoped memory-cell
- Calling a parameter without parameter returns the contents of the memory-cell
- Use `parameterize` to overwrite the memory-cell

Dynamic binding: controlled globals

> We can define different globals in different contexts.

```racket
(define buff (open-output-string))
(parameterize ([current-output-port buff])
  ; In this context, the standard output is a string buffer.
  (display "hello world!"))
(check-equal? (get-output-string buff) "hello world!")
```

Racket uses parameters to allow extending the behavior of many features:

- command line parameters
- standard output stream (known as a port)
- formating options (eg, default implementation to print structures)

Dynamic binding: making code testable

> Consider an excerpt of Homework 5. We would like to be able to test each function independently. How?

```
(define (s:eval-exp mem env exp)
  (define (on-app mem env exp)
    ;; ...
    ;; Eb \Downarrow Eb vb
    (s:eval-term mem3 Eb (s:lambda-body lam)))
  (cond
    ;; ...
    [(s:apply? exp) (on-app mem env exp)]

(define (s:eval-term mem env term)
  (cond
    ; ...
    [else (s:eval-exp mem env term)]))
```

Dynamic binding: making code testable

- In Homework 4, we added a function parameter to test `r:eval` independently from `r:subst`.
- This extra function parameter was confusing to some students.
- This choice made the function interface more verbose than needed.
- More arguments, more chance of mistakes! Do we call `subst` or `s:subst`?

How can we use dynamic binding

to improve the testing design of `r:eval`?

Dynamic binding: making code testable

- Create a parameter per global function that you want to make testable
- Internal calls should target the *parameter* and not the global variable

Before

```
(define (r:eval subst exp)
  (cond
    [(r:value? exp) exp]
    [(r:apply? exp)
     (define vf
       (r:eval subst (r:apply-func exp)))
     (define va
       (r:eval subst (r:apply-arg1 exp)))
     (define x (r:lambda-param1 vf))
     (define eb (r:lambda-body1 vf))
     (r:eval subst (subst eb x va))]))
```

Dynamic binding: making code testable

- Create a parameter per global function that you want to make testable
- Internal calls should target the *parameter* and not the global variable

Before

```
(define (r:eval subst exp)
  (cond
    [(r:value? exp) exp]
    [(r:apply? exp)
     (define vf
       (r:eval subst (r:apply-func exp)))
     (define va
       (r:eval subst (r:apply-arg1 exp)))
     (define x (r:lambda-param1 vf))
     (define eb (r:lambda-body1 vf))
     (r:eval subst (subst eb x va))]))
```

After

```
(define r:subst-impl
    (make-parameter r:subst))
(define (r:eval exp)
  (cond
    [(r:value? exp) exp]
    [(r:apply? exp)
     (define vf (r:eval (r:apply-func exp)))
     (define va (r:eval (r:apply-arg1 exp)))
     (define x (r:lambda-param1 vf))
     (define eb (r:lambda-body1 vf))
     (r:eval ((r:subst-impl) eb x va))]))
```

Dynamic binding: making code testable

> Consider an excerpt of Homework 5. We would like to be able to test each function independently. How?

```
(define (s:eval-exp mem env exp)
  (define (on-app mem env exp)
    ; ...
    ((s:eval-term-impl) mem3 Eb (s:lambda-body lam)))
  (cond ; ...
    [(s:apply? exp) (on-app mem env exp)]
(define s:eval-exp-impl (make-parameters s:eval-exp))

(define (s:eval-term mem env term)
  (cond ; ...
    [else ((s:eval-exp-impl) mem env term)]))
(define s:eval-term-impl (make-parameters s:eval-term))
```

Dynamic binding: making code testable

- Usage example:

```
(parameterize ([s:eval-expr-impl (lambda (mem env expr) (s:number 10))])
  ; Now x is evaluated to (s:number 10) and y evaluates to (s:number 10)
  (eval-term? '[x y] 10))
```

We can test eval-term without implementing eval-exp!

This testing technique is known as *mocking*.

Continuations

What is a continuation?

A technique to abstract control flow. It reifies an execution point as a pair that consists of:

- the program state (eg, the environment)
- the remaining code to run (eg, the term)

Used to encode

- **exceptions**
- **generators**
- coroutines (lightweight threads)

How can we represent continuations?

- continuation-passing style (inversion of control)
- first-class construct (Racket)

Continuation-passing style (CPS)

Q: How do we abstract computation?

Continuation-passing style (CPS)

Q: How do we abstract computation?

A: Inversion of control

> Hollywood principle: Don't call us, we'll call you.

- the objective is to have control over where a function returns to (its continuation)
- make *returning a value* a function call

Direct style

```
(define (f x)
  (+ x 2))
```

CPS

```
(define (f x ret)
  (ret (+ x 2)))
```

Where have we seen CPS?

Remember when we implemented the tail-recursive optimization?

Before

```
(define (map f l)
  (cond [(empty? l) l]
        [else (cons (f (first l)) (map f (rest l)))]))
```

After

```
(define (map f l)
  (define (map-iter l accum)
    (cond [(empty? l) (accum l)]
          [else (map-iter (rest l) (lambda (x) (accum (cons (f (first l)) x))))]))
  (map-iter l (lambda (x) x)))
```

Function `map-iter` is the CPS-version of `map`!

Encoding exceptions with CPS

```
(define (safe-/ x y)
  (lambda (ok err)
    (cond [(= 0 y) (err 'division-by-zero)]
          [else (ok (/ x y))])))
```

Example 1

```
; Print to standard-output if OK and throw an exception if not
((safe-/ 2 1) display error)
; error: division-by-zero
((safe-/ 2 0) display error)
```

Example 2

> How can we chain two divisions together?

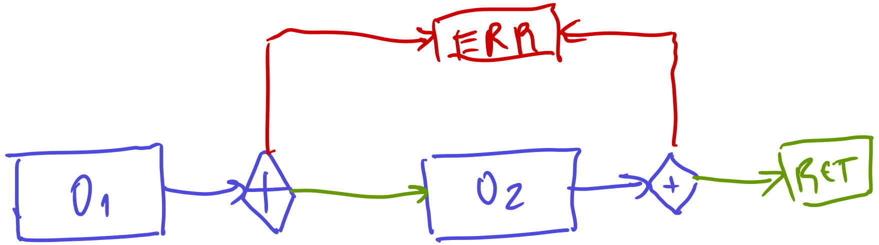
```
(/ (/ 10 2) 3)
```

Exceptions Monadic+CPS

```
; Returns x via the return function
(define (return x)
  (lambda (ret err)
    (ret x)))
; Returns x via the error function
(define (raise x)
  (lambda (ret err)
    (err x)))
; Monadic-bind on CPS-style code
(define (cps-bind o1 o2)
  (lambda (ret err)
    (o1 (lambda (res) ((o2 res) ret err)) err)))
; The try-catch operation
(define (try o1 o2)
  (lambda (ret err)
    (o1 ret (lambda (res) ((o2 res) ret err)))))
```

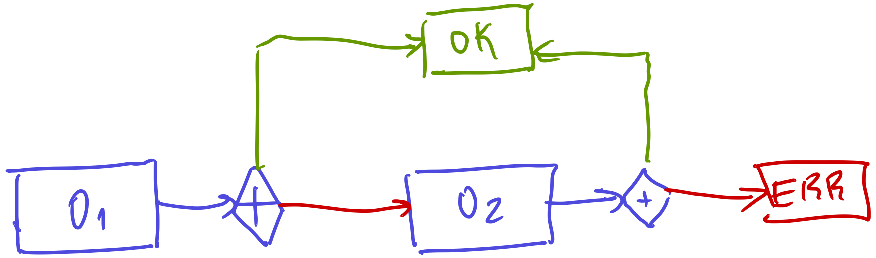
Bind

`bind` runs `o1` and the ok-continuation of `o1` is running `o2`



Try

`try` runs `o1` and the error-continuation is running `o2`



Revisiting safe-division with monadic API

> Thanks to functional programming and monads, we can easily design `try-catch` on top of a regular computation.

```
(define (&/ x y)
  (cond [(= 0 y) (raise 'division-by-zero)]
        [else (return (/ x y))]))
```

Examples

```
; 1. Run a division by zero and get an exception
(run? (&/ 1 0) (cons 'error 'division-by-zero))
; 2. Run a division by zero and use try-catch to return OK
(run?
  (try
    (&/ 1 0)
    (lambda (err) (return 10)))
  (cons 'ok 10))
; 3. Use bind in a more intricate computation
(run?
  (do
    x ← (&/ 3 4)
    (try
      (&/ x 0)
      (lambda (err) (return 10))))
  (cons 'ok 10))
```

First-class support continuations in Racket

Inversion of control

> `(call/cc f)` captures the surrounding code as a **continuation**, and passes that continuation to function `f`.

```
(+ 1 2 (call/cc f) 4 5)
```

becomes

```
(f (lambda (x) (+ 1 2 x 4 5)))
```

Recommended reading

- Many examples using call/cc

Yield: abstracting lazy evaluation

`yield` allows generalizing returning a finite stream of values (rather than just one). `yield` actually returns a value, so the caller can interact with the caller. In the following example, `yield` allows processing multiple files ensuring the garbage collector does not load everything to memory eagerly.

```
# source: https://github.com/cogumbreiro/apisan/blob/master/analyzer/apisan/parse/explorer.py
def parse_file(filename):
  # ...
  for root in xml:
      tree = ExecTree(ExecNode(root, resolver=resolver)) # load a possibly big file
      yield tree
      del tree # garbage collect the memory
## User code
for xml in parse_file(somefile):
  handle(xml) # handle the xml object
```

Implementing `yield`

Let us implement `yield` in Racket!

- Yield: Mainstream Delimited Continuations. TPDC. 2011

Papers are still being published in top Programming Language conferences on this subject:

- Theory and Practice of Coroutines with Snapshots. ECOOP. 2018

Yield summary

1. Run a CPS computation normally until `(yield x)`
2. The execution of `(yield x)` should suspend the current execution
3. There must exist an execution context that can run suspendable computations

Implementation

Yield is a regular CPS-monadic operation but it returns a suspended object, rather than using `ok` or `err`.

```
(struct susp (value ok) #:transparent)

(define (yield v)
  (lambda (ok err) (susp v ok)))

(define (resume f s)
  ((susp-ok s) (f (susp-value s))))
```

(Demo...)

Example

```
(define prog1
  (do
    (return "ignored value")
    x1 ← (yield 1)
    ;(raise 'foo)
    x2 ← (yield 2)
    (return (cons x1 x2))))
```

How do we catch exception in Racket?

> We must use the `with-handler` construct that takes the exception type, and the code that is run when the exception is raised.

```
#lang racket
(require rackunit)
(define (f)
  (define (on-err e)
    ; Instead of returning what we were doing, just return #f
    #f)
  (with-handler ([exn:fail:contract:divide-by-zero? (lambda (e) #f)])
    (/ 1 0)))
; The handler is called and the final result is #f
(check-false (f))
```