CS450

Structure of Higher Level Languages

Lecture 22: Error monad / CPS monad

Tiago Cogumbreiro

Evaluating expressions

Handling errors

```
(: r:eval-exp (-> r:expression (Option Real)))
(define (r:eval-exp exp)
  (match exp
    ; If it's a number, return that number
    [(r:number v) v]
    ; If it's a function with 2 arguments
    [(r:apply (r:variable f) arg1 arg2)
      (define func (r:eval-builtin f))
      (define a1 (r:eval-exp arg1))
      (cond [(false? a1) #f]
            [else
              (define a2 (r:eval-exp arg2))
              (cond [(false? a2) #f]
                [else
                  (func a1 a2)])])]))
```

UMass Boston

Error handling API

How can we abstract this pattern?

```
(define a1 (r:eval-exp arg1))
(cond
  [(false? a1) #f]
  [else
    (define a2 (r:eval-exp arg2))
    (cond
      [(false? a2) #f]
      [else (func a1 a2)])])
```

How can we abstract this pattern?

```
(define a1 (r:eval-exp arg1))
(cond
  [(false? a1) #f]
  [else
    (define a2 (r:eval-exp arg2))
    (cond
      [(false? a2) #f]
      [else (func a1 a2)])])
```

Refactoring

```
(define (handle-err res kont)
  (cond
    [(false? res) #f]
    [else (kont res)]))
```

Rewriting our code with `handle-err`

(Demo...)

Rewriting our code with `handle-err`

(Demo...)

```
(handle-err (r:eval-exp arg1)
  (lambda ([a1 : Real])
    (handle-err (r:eval-exp arg2)
      (lambda ([a2 : Real]) : (Option Real)
        (func a1 a2)))))
```

Example 3

```
(r:eval-exp (r:apply (r:variable 'modulo) (list (r:number 1) (r:number 0))))
; application: not a procedure;
;  expected a procedure that can be applied to arguments
;   given: #f
; [,bt for context]
```

Let us revisit `r:eval`

(Demo…)

Let us revisit `r:eval`

(Demo...)

```
(handle-err (r:eval-exp (r:apply-func exp))
  (lambda (func)
    (handle-err (r:eval-exp (first (r:apply-args exp)))
      (lambda (arg1)
        (handle-err (r:eval-exp (second (r:apply-args exp)))
          (lambda (arg2)
            (func arg1 arg2)))))))
```

Where have we seen this before?

Let us revisit `r:eval`

(Demo...)

```
(handle-err (r:eval-exp (r:apply-func exp))
  (lambda (func)
    (handle-err (r:eval-exp (first (r:apply-args exp)))
      (lambda (arg1)
        (handle-err (r:eval-exp (second (r:apply-args exp)))
          (lambda (arg2)
            (func arg1 arg2)))))))
```

Where have we seen this before?

Monads!

Handling errors with monads

Monads

A general functional pattern that abstracts assignment and control flow

- Monads are not just for handling state
- Monads were introduced in Haskell by Philip Wadler in 1990

The monadic interface

- **Bind:** combines two effectful operations o_1 and o_2. Operation o_1 produces a value that is consumed by operation o_2.

```
(define (handle-err res kont) (cond [(false? res) #f] [else (kont res)])) ; For err
```

- **Pure:** Converts a pure value to a monadic operation, which can then be chained with `bind`.

```
(define (pure e) e)  ; For err
```

Re-implementing the do-notation

- Let us copy-paste our macro and replace `eff-bind` by `handle-err`.

```
(define-syntax do
  (syntax-rules (<-)
    ; Only one monadic-op, return it
    [(_ mexp) mexp]
    ; A binding operation
    [(_ var : ty <- mexp rest ...) (handle-err mexp (lambda ([var : ty]) (do rest ...)))]
    ; No binding operator, just ignore the return value
    [(_ mexp rest ...)        (handle-err mexp (lambda (_) (do rest ...)))]))
```

Rewriting `r:eval`

(Demo...)

Rewriting `r:eval`

(Demo...)

```
(define (r:eval-exp exp)
  (match exp
    ; If it's a number, return that number
    [(r:number v) v]
    ; If it's a function with 2 arguments
    [(r:apply (r:variable f) arg1 arg2)
      (define func (r:eval-builtin f))
      (do
        a1 : Real <- (r:eval-exp arg1)
        a2 : Real <- (r:eval-exp arg2)
        (func a1 a2)
      )]))
```

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-type (CPS Ok Error Result)
  (-> (-> Ok Result) (-> Error Result) Result))
(: safe-/ (All [Result] (-> Real Real (CPS Real Symbol Result))))
(define (safe-/ x y)
  (lambda (ok err)
    (cond [(= 0 y) (err 'division-by-zero)]
          [else (ok (/ x y))])))
```

Example 1

```
((safe-/ 2 1) displayln error)
((safe-/ 2 0) displayln error)
```

Example 2

> How can we chain two divisions together?

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

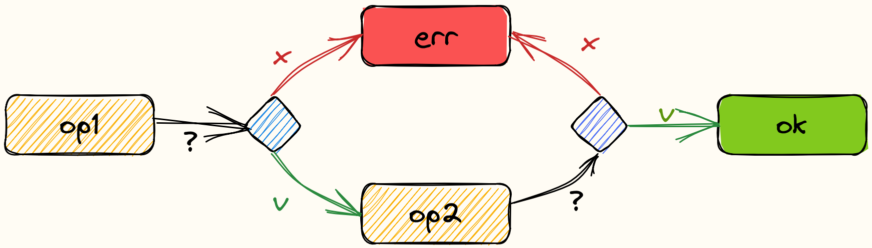
Monadic

Continuation-Passing Style

Exceptions Monadic+CPS

Bind

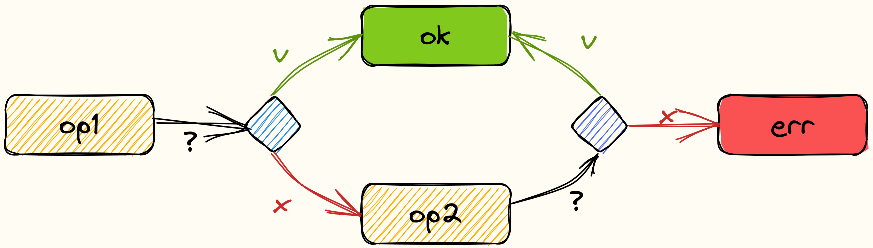
`bind` runs `o1` and the ok-continuation of `o1` is running `o2`



1. Try to run `o1`
2. If `o1` raises an exception, send it to `err`
3. Otherwise, send result of `o1` to `o2` and try to run it
4. If `o2` raises an exception, send it to `err`
5. Otherwise, send result of `o2` to `ok`

Try

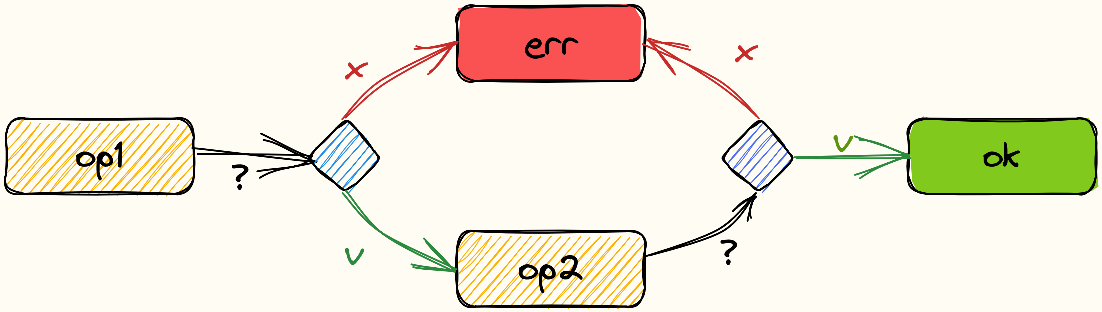
`try` runs `o1` and the error-continuation is running `o2`



1. Try to run `o1`
2. If `o1` resumes successfully, send it to `ok`
3. Otherwise, send `o1` exception to `o2` (ie, the "catch" block)
4. If the "catch" block sends an exception, send it to `err`
5. Otherwise, send the result of `o2` to `ok`

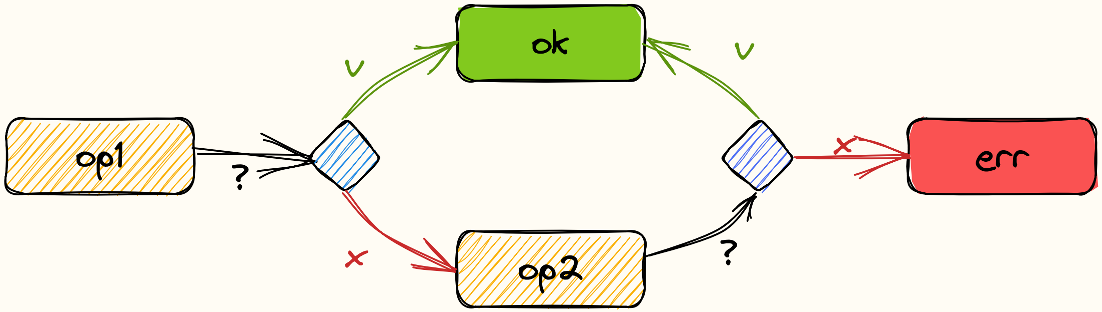
Specifying sequencing

```
(: cps-bind
  (All [Ok1 Ok2 Error Result]
    (->
      ; The type of the first continuation
      (CPS Ok1 Error Result)
      ; Given the result of the first, return a continuation
      (-> Ok1 (CPS Ok2 Error Result))
      ; The second continuation
      (CPS Ok2 Error Result))))
```



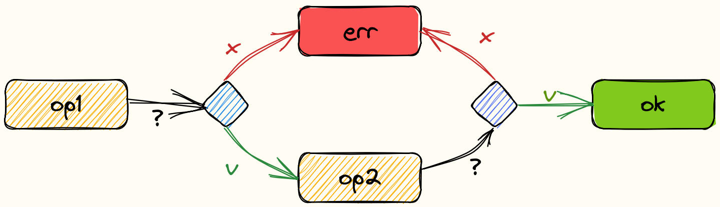
Specifying the try/catch

```
(: try
  (All [Ok Error1 Error2 Result]
    (->
      ; The type of the first continuation
      (CPS Ok Error1 Result)
      ; Given the result of the first, return a continuation
      (-> Error1 (CPS Ok Error2 Result))
      ; The second continuation
      (CPS Ok Error2 Result))))
```



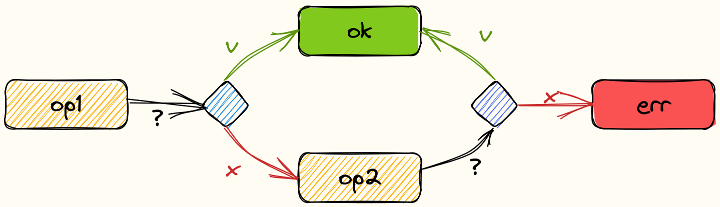
Implementing sequencing

```
(define (cps-bind o1 o2)
  (lambda ([ret : (-> Ok2 Result)] [err : (-> Error Result)])
    ; Run the first operation
    (o1
      (lambda ([res : Ok1])
        ; If the operation is successful
        (
          ; Build the second CPS operation
          (o2 res)
          ; And pass the original ret/err pairs
          ret
          err
        )
      )
      ; Otherwise, pass the same error
      err)))
```



Implementing try/catch

```
(define (try o1 o2)
  (lambda ([ret : (-> Ok Result)] [err : (-> Error2 Result)])
    ; Try to run o1
    (o1
      ; If successful, pass it to ret
      ret
      ; Otherwise, call the catch block
      (lambda ([res : Error1])
        (
          ; Build the "catch" block
          (o2 res)
          ; Pass the parent return channel
          ret
          ; Pass the parent exception channel
          err
        )
      )
    )
  )
)
```

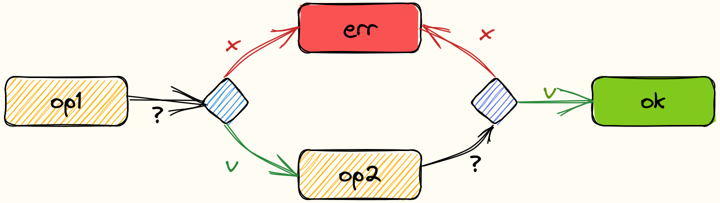


Exceptions Monadic+CPS

```
; 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)))))
```

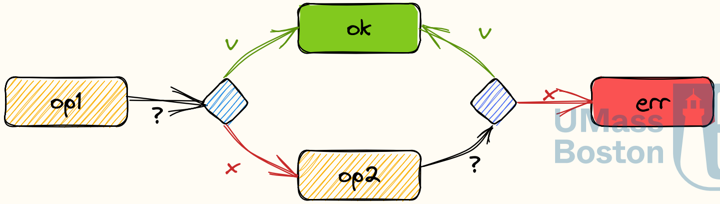
Bind

`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))
```

Exceptions in Racket

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
(define (on-err e)
  ; Instead of returning what we were doing, just return #f
  #f)
(with-handlers ([exn:fail:contract:divide-by-zero? on-err])
  (/ 1 0))
```

First-class continuations in Racket

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

Another way to write streams

(Or, returning streams of values)

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.p
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 s)
  ((susp-ok s) (void)))
```

(Demo...)

Monadic List Comprehension

Monad: List comprehension

List comprehension is a mathematical notation to succinctly describe the members of the list.

$$\left[(x, y) \mid x \leftarrow [1, 2]; y \leftarrow [3, 4]\right] = \left[(1, 3), (1, 4), (2, 3)(2, 4)\right]$$

```
(define lst
  (do
    x <- (list 1 2)
    y <- (list 3 4)
    (list-pure (cons x y))))
; Result
(check-equal? lst (list (cons 1 3) (cons 1 4) (cons 2 3) (cons 2 4)))
```

Designing the list monad

The `join` operation

Spec

```
(check-equal? (join (list (list 1 2)))
  (list 1 2))
(check-equal? (join (list (list 1) (list 2)))
  (list 1 2))
(check-equal? (join (list (list 1 2) (list 3)))
  (list 1 2 3))
```

Designing the list monad

The `join` operation

Spec

```
(check-equal? (join (list (list 1 2)))
  (list 1 2))
(check-equal? (join (list (list 1) (list 2)))
  (list 1 2))
(check-equal? (join (list (list 1 2) (list 3)))
  (list 1 2 3))
```

Solution

```
(define (join elems)
  (foldr append empty elems))
```

Designing the list monad

```
(define (list-pure x) (list x))

(define (list-bind op1 op2)
  (join (map op2 op1)))
```

Re-implementing the do-notation

- Let us copy-paste our macro and replace `bind` by `list-bind`.

```
(define-syntax do
  (syntax-rules (<-)
    ; Only one monadic-op, return it
    [(_ mexp) mexp]
    ; A binding operation
    [(_ var <- mexp rest ...) (list-bind mexp (lambda (var) (do rest ...)))]
    ; No binding operator, just ignore the return value
    [(_ mexp rest ...)        (list-bind mexp (lambda (_) (do rest ...)))]))
```

Desugaring list comprehension

```
(define lst
  (do
    x <- (list 1 2)
    y <- (list 3 4)
    (pure (cons x y))))
; =
(define lst
  (list-bind (list 1 2)
    (lambda (x)
      (list-bind (list 3 4)
        (lambda (y)
          (list-pure (cons x y)))))))
```

```
(join
  (map
    (lambda (x)
      (join (map (lambda (y) (list (cons x y))) (list 3 4))))
    (list 1 2)))
; =
(join
  (map
    (lambda (x) (join (list (list (cons x 3)) (list (cons x 4)))))
    (list 1 2)))
; =
(join
  (map
    (lambda (x) (list (cons x 3) (cons x 4)))
    (list 1 2)))
; =
 (join (list (list (cons 1 3) (cons 1 4)) (list (cons 2 3) (cons 2 4))))
; =
(list (cons 1 3) (cons 1 4) (cons 2 3) (cons 2 4))
```

Examples

Example 1

```
(check-equal? (list-bind (lambda (x) (list x x)) (list 1 2 3))
```

Examples

Example 1

```
(check-equal? (list-bind (lambda (x) (list x x)) (list 1 2 3))

  (list 1 1 2 2 3 3))
```

Example 2

```
(check-equal? (do x <- (list 1 2) (list (* x 10) (+ x 2) (- x 1)))
```

Examples

Example 1

```
(check-equal? (list-bind (lambda (x) (list x x)) (list 1 2 3))
   (list 1 1 2 2 3 3))
```

Example 2

```
(check-equal? (do x <- (list 1 2) (list (* x 10) (+ x 2) (- x 1)))
  (list 10 3 0 20 4 1))
```

Example 3

```
(check-equal? (list-bind (lambda (x) (list)) (list 1 2 3))
```

Examples

Example 1

```
(check-equal? (list-bind (lambda (x) (list x x)) (list 1 2 3))
   (list 1 1 2 2 3 3))
```

Example 2

```
(check-equal? (do x <- (list 1 2) (list (* x 10) (+ x 2) (- x 1)))
  (list 10 3 0 20 4 1))
```

Example 3

```
(check-equal? (list-bind (lambda (x) (list)) (list 1 2 3))
  (list))
```

Examples

Example 4

```
(check-equal? (do x <- (list 1 2 3 4) (if (even? x) (pure x) empty))
```

Examples

Example 4

```
(check-equal? (do x <- (list 1 2 3 4) (if (even? x) (pure x) empty))
  (list 1 3))
```

$$\left[x \mid x \leftarrow [1,2,3,4] \text{ if even?}(x)\right] = [1,3]$$