CS450

Structure of Higher Level Languages

Lecture 21: Loops and error monad

Tiago Cogumbreiro

Press arrow keys ← → to change slides.

Writing recursive code with monads

Version 1:

Using eff-bind

Revisiting copy (types)

```
(: copy
  (All [T]
    (->
      (Listof T)
      (Listof T)
    )
  )
)
```

```
(: eff-copy
  (All [State T]
    (->
      ; Given a list of effectful ops
      (Listof (eff-op State T))
      ; Return an effectful operation that produces
      ; a list of results
      (eff-op State (Listof T))
    )
  )
)
```

Revisiting copy (code)

```
(define (copy l)
  (match l
    [(list) (list)]
    [(list h l ...)
      (define result (copy l))
      (cons h result)
    ]
  )
)
```

```
(define (eff-copy l)
  (match l
    [(list)
      ; Return (list) with eff-pure
      (eff-pure (list))]
    [(list h l ...)
      ; Similar to: (define h-value h)
      (eff-bind h
        (lambda ([h-value : T])
          ; Similar to: (define result (eff-copy l))
          (eff-bind (eff-copy l)
            (lambda ([result : (Listof T)])
              ; Return: (cons h-value result)
              (eff-pure (cons h-value result))
            ))))]))
```

Summary of version 1

- Call `eff-pure` to return expression
- In recursive step:
  1. Evaluate `h` first with `eff-bind`
  2. Evaluate recursive call second with eff-bind
  3. Handle `h-value` and `result`
  4. Annotate the types of the lambdas
  5. Annotate the return type of the **inner-most** lambda

```
(define (eff-copy l)
  (match l
    [(list) ; return with eff-pure
      (eff-pure (list))]
    [(list h l ...)
      (eff-bind h
        (lambda ([h-value : T])
          (eff-bind (eff-copy l)
            (lambda ([result : (Listof T)])
              (eff-pure (cons h-value result))
            ))))]))
```

Version 2:

Using do

Revisiting copy (code) / do-notation

```
(define (copy l)
  (match l
    [(list) (list)]
    [(list h l ...)
      (define result (copy l))
      (cons h result)
    ]
  )
)
```

```
(define (eff-copy l)
  (match l
    [(list) (eff-pure (list))]
    [(list h l ...)
      (do
        h-value : T <- h
        result : (Listof T) <- (eff-copy l)
        (ann (eff-pure (cons h-value result))
          ; you must annotate the return type
          (eff-op State (Listof T)))
      )]))
```

Summary of version 2

- Use `(do ...)` instead of `(eff-bind ...)`
- A single `(do ...)` block can capture multiple nested `(eff-bind ...)` blocks
- Each `(eff-bind e (lambda ([x : T]) ...)` becomes an `x : T <- e` where `e` has type `(eff-op State T)`
- When there are multiple binds, annotate the return value of the last bind, the type must be `(eff-op State T)`

```
(define (eff-copy l)
  (match l
    [(list) (eff-pure (list))]
    [(list h l ...)
      (do
        h-value : T <- h
        result : (Listof T) <- (eff-copy l)
        (ann (eff-pure (cons h-value result))
             ; you must annotate the return type
             (eff-op State (Listof T))))
      )]))
```

Version 3:

Using an iterative loop

Rethink copy in iterative-style

```
(define (copy l)
  (: loop ; In TypedRacket we must
          ; supply type signature
    (->
      (Listof T) ; Accum type
      (Listof T) ; List type
      (Listof T) ; Accum type
    )
  )
  (define (loop accum l)
    (match l
      [(list) (reverse accum)]
      [(list h l ...)
        (loop (cons h accum) l)
      ]))
  (loop (list) l))
```

```
# Pseudo-code
accum = []
for x in l:
  accum.insert(0, x)
return list(reversed(l))
```

Rethink copy in iterative-style

```
(define (copy l)
  (: loop ... )
  (define (loop accum l)
    (match l
      [(list) (reverse accum)]
      [(list h l ...)
        (loop (cons h accum)  l)
      ]))
  (loop (list) l))
```

```
(define (eff-copy l)
  (: loop ... )
  (define (loop accum l)
    (match l
      [(list) (eff-pure (reverse accum))]
      [(list h l ...)
        (do
          h-value : T <- h ; Must evaluate h
          (loop (cons h-value accum) l)
        )]))
  (loop (list) l))
```

Example: sum

Sum the results of a list of monads

```
(: sum
  (All [State]
    (->
      ; Given a list of effectful ops
      (Listof (eff-op State Real))
      ; Add all of the results
      (eff-op State Real)
    )
  )
)
```

UMass Boston

Sum the results of a list of monads

Without accumulator = more typing information

```
(define (sum l)
  (match l
    [(list) (eff-pure 0)]
    [(list h l ...)
      (eff-bind h
        (lambda ([h-val : Real]) : (eff-op State Real)
          (eff-bind (sum l)
            (lambda ([l-val : Real]) : (eff-op State Real)
              (eff-pure (+ h-val l-val))))))]))
```

Sum the results of a list of monads

Without accumulator = more typing information

```
(define (sum l)
  (match l
    [(list) (eff-pure 0)]
    [(list h l ...)
      (do
        h-val : Real <- h
        l-val : Real <- (sum l)
        (ann (eff-pure (+ h-val l-val)) (eff-op State Real)))]))
```

Without the annotation you would get this error, notice the occurrence of `Any`:

```
Argument 2:
  Expected: (-> Input (-> State (eff State Output)))
  Given:    (-> Real (-> Any (eff Any Real)))
```

Sum the results of a list of monads

With accumulator

```
(define (sum l)
  (: sum-iter (-> Real (Listof (eff-op State Real)) (eff-op State Real)))
  (define (sum-iter accum l)
    (match l
      [(list) (eff-pure accum)]
      [(list h l ...)
        (do
          h-val : Real <- h
          (sum-iter (+ h-val accum) l)
        )
      ]
    )
  )
  (sum-iter 0 l))
```

Notes when writing recursive code

- Accumulator functions are preferred, as the typing information is more obvious
- Accumulators of type list may need to be reversed before returning to satisfy ordering
- Writing more "natural" recursive patterns leads to typing complications
- You can use `(ann expression type)` when the type-checker complains about `Any`
- **Compare the two solutions of `sum` presented in this class with your solution used in HW2** this gives you the blue-print to solve the last 2 exercises of the homework assignment.

UMass Boston

Error handling

Recall our interpreter from HW3

```
(define (r:eval-builtin sym)
  (cond [(equal? sym '+) +]
        [(equal? sym '*) *]
        [(equal? sym '-) -]
        [(equal? sym '/) /]
        [else #f]))

(define (r:eval-exp exp)
  (cond
    [(r:number? exp) (r:number-value exp)]
    [(r:variable? exp) (r:eval-builtin (r:variable-name exp))]
    [(r:apply? exp)
     ((r:eval-exp (r:apply-func exp))
      (r:eval-exp (first (r:apply-args exp)))
      (r:eval-exp (second (r:apply-args exp))))]
    [else (error "Unknown expression:" exp)]))
```

Consider the following example

- What happens if we run this example?

```
(r:eval-exp 10)
```

Consider the following example

- What happens if we run this example?

```
(r:eval-exp 10)
```

```
; Unknown expression: 10
;   context...:
```

The caller should be passing an AST, not a number!

- We should be using contracts to avoid this kind of error!

Consider the following example

- What happens if the user tries to divide a number by zero?

```
(r:eval-exp (r:apply (r:variable '/) (list (r:number 1) (r:number 0))))
```

Consider the following example

- What happens if the user tries to divide a number by zero?

```
(r:eval-exp (r:apply (r:variable '/) (list (r:number 1) (r:number 0))))
```

```
; /: division by zero
;   context...:
```

Is this considered an error?

How can we solve this problem?

How can we solve this problem?

What does the error mean?

> Is this a user error? Or is this an implementation error?

How can we solve this problem?

What does the error mean?

> Is this a user error? Or is this an implementation error?

Is it an implementation problem?

> **Implementation errors should be loud!** We want our code to crash during testing. This family of errors could correspond to a bug, or, more importantly, to a misunderstanding between the developer and the client! Using the exceptions model of our client is a big plus, as we get stack trace information, among other niceties.

How can we solve this problem?

What does the error mean?

> Is this a user error? Or is this an implementation error?

Is it an implementation problem?

> **Implementation errors should be loud!** We want our code to crash during testing. This family of errors could correspond to a bug, or, more importantly, to a misunderstanding between the developer and the client! Using the exceptions model of our client is a big plus, as we get stack trace information, among other niceties.

Is it a user error?

> User errors must be handled **gracefully** and ***cannot*** crash our application. User errors must also not reveal the internal state of the code (**no stack traces!**), as such information can pose a security threat.

Handling run-time errors

Solving the division-by-zero error

1. We can implement a safe-division that returns a special return value
2. We can let Racket crash and catch the exception

Implementing safe division

- Implement a safe-division that returns a special return value

Implementing safe division

- Implement a safe-division that returns a special return value

```
(define (safe-/ x y)
  (cond [(= y 0) #f]
        [else (/ x y)]))
```

Is this enough?

Is this enough?

```
(r:eval-exp
  (r:apply
    (r:variable '+)
    (list
      (r:apply (r:variable '/) (list (r:number 1) (r:number 0)))
      (r:number 10))))
; +: contract violation
;   expected: number?
;   given: #f
;   argument position: 1st
; [,bt for context]
```

We still need to rewrite `r:eval-exp` to handle `#f`

UMass Boston

Solving apply

- (Demo...)

Solving apply

- (Demo...)

```
(: 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)])])]))
```