CS450

Structure of Higher Level Languages

Lecture 29: Refactoring errors; monads

Tiago Cogumbreiro
Today we will learn about...

- Errors in our interpreter: user errors versus implementation errors
- Handling implementation errors
- Refactoring our interpreter
- Handling errors with a monadic interface
- Monadic list comprehension
Recall our interpreter from HW3

\begin{verbatim}
(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)])
\end{verbatim}
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
(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`
Solving apply

(Demo...)
Solving apply

(Demo...)

```
(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)
    (define arg1 (r:eval-exp (first (r:apply-args exp))))
    (cond
     [(false? arg1) arg1]
     [else
      (define arg2 (r:eval-exp (second (r:apply-args exp))))
      (cond
       [(false? arg2) arg2]
       [else ((r:eval-exp (r:apply-func exp)) arg1 arg2)]])]
   [else (error "Unknown expression:" exp)])
```
Error handling API
How can we abstract this pattern?

```
(define arg1 (r:eval-exp (first (r:apply-args exp))))
(cond
  [(false? arg1) arg1]
  [else
   (define arg2 (r:eval-exp (second (r:apply-args exp))))
   (cond
     [(false? arg2) arg2]
     [else ((r:eval-exp (r:apply-func exp)) arg1 arg2)])])
```
How can we abstract this pattern?

```
(define arg1 (r:eval-exp (first (r:apply-args exp))))
(cond
  [(false? arg1) arg1]
  [else
   (define arg2 (r:eval-exp (second (r:apply-args exp))))
   (cond
    [(false? arg2) arg2]
    [else ((r:eval-exp (r:apply-func exp)) arg1 arg2)])])
```

Refactoring

```
(define (handle-err res kont)
  (cond
   [(false? res) res]
   [else (kont res)])
```
Rewriting our code with \texttt{handle-err}

(Demo...)
Rewriting our code with `handle-err`

(Demo...)

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

```lisp
(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...)

```scheme
(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 $o_2$.

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

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

  ```scheme
  (define (pure e) e); For err
  ```
Re-implementing the do-notation

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

```scheme
(define-syntax do
  (syntax-rules ()
    [(mexp) mexp] ; Only one monadic-op, return it
    [(var <- mexp rest ...) (handle-err mexp (lambda (var) (do rest ...)))] ; A binding operation
    [(mexp rest ...) (handle-err mexp (lambda (_) (do rest ...)))]))
```
Rewriting `r:eval-builtIn`

(Demo...)
Rewriting `r:eval-builtin`

(Demo...)

```clojure
(do
  func  ← (r:eval-exp (r:apply-func exp))
  arg1  ← (r:eval-exp (first (r:apply-args exp)))
  arg2  ← (r:eval-exp (second (r:apply-args exp)))
  (func arg1 arg2))
```
Monadic List Comprehension
Monad: List comprehension

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

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

(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

\[
\text{(define \textit{lst}} \\
\text{(do} \\
\quad \text{x } \leftarrow \text{(list 1 2)} \\
\quad \text{y } \leftarrow \text{(list 3 4)} \\
\quad \text{(pure (\textit{cons} x y)))}} \\
\text{)}
\]

; = 

\[
\text{(define \textit{lst}} \\
\text{(\textit{list-bind} (\textit{list} 1 2) \\
\quad \text{(lambda} (x) \\
\quad \quad \text{((\textit{list-bind} (\textit{list} 3 4) \\
\quad \quad \quad \text{(\textit{lambda} (y) \\
\quad \quad \quad \quad \text{(\textit{list-pure} (\textit{cons} x y))))))))}} \\
\text{)}}
\]
(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)))
               (list 1 1 2 2 3 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))
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))
Example 4

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

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