CS450

Structure of Higher Level Languages

Lecture 19: Monadic error/list; generics; parameters

Tiago Cogumbreiro
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?

\[(\text{r:eval-exp} (\text{r:apply} (\text{r:variable} '/) (\text{list} (\text{r:number} 1) (\text{r:number} 0))))\]
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?

We still need to rewrite \texttt{r:eval-exp} to handle \#f.
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)])

CS450 ☽ Monadic error/list; generics; parameters ☽ Lecture19 ☽ Tiago Cogumbreiro
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 `handle-err`.

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

(Demo...)

```
(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 \texttt{r:eval}

(Demo...)

(Monadic error/list; generics; parameters)
Let us revisit \texttt{r:eval}.

(Demo...)

\begin{verbatim}
(\texttt{handle-err (r:eval-exp (r:apply-func \texttt{exp}))}
 \texttt{(lambda (func)}
 \texttt{(handle-err (r:eval-exp (first (r:apply-args \texttt{exp})))}
 \texttt{(lambda (arg1)}
 \texttt{(handle-err (r:eval-exp (second (r:apply-args \texttt{exp})))}
 \texttt{(lambda (arg2)}
 \texttt{(func arg1 arg2))))))))
\end{verbatim}

Where have we seen this before?
Let us revisit \texttt{r:eval}

(Demo...)

\begin{verbatim}
(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)))))
)
\end{verbatim}

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 $o_2$.

  \[
  \text{(define (handle-err res kont) (cond [(false? res) res] [else (kont res)])) \; \text{For err}}
  \]

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

  \[
  \text{(define (pure e) e) \; \text{For err}}
  \]
Re-implementing the do-notation

Let us copy-paste our macro and replace \texttt{bind} by \texttt{handle-err}.

\begin{verbatim}
(define-syntax do
  (syntax-rules (↔)
    ; Only one monadic-op, return it
    [(_ mexp) mexp]
    ; A binding operation
    [(_ var ↔ mexp rest ...) (handle-err mexp (lambda (var) (do rest ...)))]
    ; No binding operator, just ignore the return value
    [(_ mexp rest ...) (handle-err mexp (lambda (_) (do rest ...)))]
  )
\end{verbatim}
Rewriting \texttt{r:eval-built-in}

(Demo...)
Rewriting `r:eval-built-in`

(Demo...)

```scheme
(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 \texttt{join} operation

Spec

\begin{verbatim}
(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))
\end{verbatim}
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))
(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`.

```scheme
(define-syntx 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 ...)))]
  )
)```
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
    (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

\[
(\text{check-equal?} \; (\text{do} \; x \leftarrow (\text{list} \; 1 \; 2 \; 3 \; 4) \; (\text{if} \; (\text{even?} \; x) \; (\text{pure} \; x) \; \text{empty})) \\
(\text{list} \; 1 \; 3))
\]

\[
[x \mid x \leftarrow [1, 2, 3, 4] \; \text{if} \; \text{even?}(x)] = [1, 3]
\]
Dynamic dispatch
(aka operator overload)

Motivation
The problem: how to unify syntax?

Three different possibilities of the same pattern

State monad

```
(define (eff-bind o1 o2)
  (lambda (h1)
    (define eff-x (o1 h1))
    (define x (eff-result eff-x))
    (define h2 (eff-state eff-x))
    (define new-op (o2 x))
    (new-op h2)))
(define (eff-pure v)
  (lambda (h) (eff h v)))
```

Error monad

```
(define (err-bind v k)
  (define arg1 v)
  (cond
    [(false? v) v]
    [else (k v)]))
(define (err-pure v) v)
```

List monad

```
(define (list-bind op1 op2)
  (join (map op2 op1)))
(define (list-pure x)
  (list x))
```
Can we do better?

Can we avoid copy-pasting our macro?
Let us study two solutions

1. Make the macro parametric
2. Use dynamic dispatch (aka operator overload)
Option 1: parametric notation

(manual dynamic dispatch)
Option 1: parametric notation

- Add a level of indirection
- Lookup a structure that holds bind and pure
- Add notation on top of that structure
The struct Monad

```
(struct monad (bind pure))
```

Redefine macro

```
(define-syntax do-with
  (syntax-rules (< pure)
    [(_ m (pure mexp)) ((monad-pure m) mexp)]
    [(_ m mexp) mexp]
    ; A binding operation
    [(_ m var < (pure mexp) rest ...) ((monad-bind m) ((monad-pure m) mexp) (lambda (var) (do-with m rest ...)))]
    [(_ m var < mexp rest ...) ((monad-bind m) mexp (lambda (var) (do-with m rest ...)))]
    ; No binding operator, just ignore the return value
    [(_ m (pure mexp) rest ...) ((monad-bind m) ((monad-pure m) mexp) (lambda (_) (do-with m rest ...)))]
    [(_ m mexp rest ...) ((monad-bind m) mexp (lambda (_) (do-with m rest ...)))]
  ))
```
Example 1

```scheme
(define list-m (monad list-bind list-pure))

(do-with list-m
  x <- (list 1 2)
  y <- (list 3 4)
  (pure (cons x y)))
```
Example 2

```scheme
(define state-m (monad eff-bind eff-pure))

(define mult
  (do-with state-m
    x ← pop
    y ← pop
    (push (* x y))))
```
Option 2:
Type-directed dynamic dispatching
Limitations

- The types of values need to be consistent
- Idea: wrap values with structs
- Use a single function `ty-bind` to perform dynamic dispatching

Implementation

```
(define (ty-bind o1 o2)
  (cond
   [(eff-op? o1) (eff-bind2 o1 o2)]
   [(optional? o1) (opt-bind o1 o2)]
   [(list? o1) (list-bind o1 o2)]))
```
Type-directed effectful operations

An effectful operations is a function that takes a state and returns an effect. Racket has no way of being able to identify that, so we need to wrap functions with a struct to mark them as effectful operations.

```
(struct eff-op (func) #:transparent)

(define/contract (eff-bind2 o1 o2)
    (→ eff-op? (→ any/c eff-op?) eff-op?)
    (eff-op (lambda (h1)
        (define/contract eff-x eff? ((eff-op-func o1) h1))
        (define x (eff-result eff-x))
        (define h2 (eff-state eff-x))
        (define/contract new-op eff-op? (o2 x))
        ((eff-op-func new-op) h2)))))
```
Type-directed effectful operation

Re-implementing the stack-machine operations. Notice that the do-notation calls \textit{ty-bind}, which in turn calls \textit{eff2-bind}.

```
(define pop2 (eff-op pop))
(define (push2 n) (eff-op (push n)))
(define mult2
  (do
    x ← pop2
    y ← pop2
    (push2 (* x y))))
```
Type-directed optional result

Optional values

```
(struct optional (data))

(define (opt-bind o1 o2)
  (cond
    [(and (optional? o1) (false? (optional-data o1))) #f]
    [else (o2 (optional-data o1))]))

(define (opt-pure x) (optional x))
```
Limitations

1. No way to implement pure.
2. If we need to add a new type, we will need to change ty-bind

```scheme
(define (ty-bind o1 o2)
  (cond [(eff-op? o1) (eff-bind2 o1 o2)]
        [(optional? o1) (opt-bind o1 o2)]
        [(list? o1) (list-bind o1 o2)]))
```
Can we do better?

Racket generics = implicit+automatic dynamic dispatching
Defining a dynamic-dispatch function

1. We use `define-generics` to declare a function that is dispatched dynamic according to the type

   *Think declaring an abstract function.*

2. We inline each version of each type inside the structure

   *Think giving a concrete implementation of an abstract function.*

```racket
(require racket/generic)
; Create a generic function that is dynamically dispatch on type ty-monad
(define-generics ty-monad
  (dyn-bind ty-monad k))

; Declare eff-op as before, but also give an instance of `dyn-bind`
(struct eff-op (op)
  #:methods gen:ty-monad
  ; Copy/paste body of eff-bind2
  [(define (dyn-bind o1 o2 ...)])]
```