CS450

Structure of Higher Level Languages

Lecture 30: Dynamic dispatching

Tiago Cogumbreiro

Today we will learn...

- Dynamic dispatching
- Manual dynamic-dispatching
- Type-directed dynamic dispatching
- Type-directed dynamic dispatching with `generic`
- Exceptions in Racket

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)
    ; Only one monadic-op, return it
    [(_ 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

```
(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

```
(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

Type-directed bind

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 `ty-bind`, which in turn calls `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`

```
(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.

```
(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) ...)])
```

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))
```