CS450

Structure of Higher Level Languages

Lecture 28: Effectful operations

Tiago Cogumbreiro

Today we will...

- Introduce a functional pattern monads
- Introduce state monads
- Introduce stack machines

Functional pattern:

Mutable state

Revisiting our reduction rules

$$\blacktriangleright_H v \Downarrow_E v \blacktriangleright_H$$

$$\frac{\blacktriangleright_{H_1} e_f \Downarrow_E (E_f, \lambda x.t_b) \;\blacktriangleright_{H_2}\; e_a \Downarrow_E v_a \;\blacktriangleright_{H_3}\; E_b \leftarrow E_f + [x := v_a] \;\blacktriangleright_{H_4} t_b \Downarrow_{E_b} v_b \blacktriangleright_{H_5}}{\blacktriangleright_{H_1} (e_f\ e_a) \Downarrow_E v_b \blacktriangleright_{H_5}}$$

Effectful computation can be divided into three categories:

- Side-effect free computation in blue
- Computation that directly produces side effect in red
- Computation that indirectly produces some side-effect in black

We are ▶ chaining ▶ effectful ▶ computations ▶, that is the variables declared on the left-hand side of ▶ should be accessible in the right-hand side.

Refactoring memory-based operations

```
;; e1 ⇓E v1
(define v1+mem1 (d:eval-exp mem env e1))
(define mem1 (eff-state v1+mem1))
(define v1 (eff-result v1+mem1))
;; E' ← E + [x := v1]
(define env2+mem2 (environ-push mem1 env y v1)
(define env2 (eff-result env2+mem2))
(define mem2 (eff-state env2+mem2))
;; e2 ⇓E' v2
(define v2+mem3 (d:eval-exp mem2 env2 e2))
(define mem3 (eff-state v2+mem3))
(define v2 (eff-result v2+mem3))
```

The memory needs to be passed along from one function call to the next. How can we refactor this code so that some function does that for us?

Refactoring evaluation of application

```
;; e1 ⇓E v1
(define* v1 (d:eval-exp* env e1))
;; E' ← E + [x := v1]
(define* env2 (environ-push* env y v1)
;; e2 ⇓E' v2
(define* v2 (d:eval-exp* env2 e2))
```

> At each step we separate the result from the state.
> Our goal is to **abstract** the memory threading, that is to refactor away this mechanic unpacking of the side effect structure.

Abstracting the state

Ideal pseudo code

> In today's class, we introduce an abstraction that allows us to achieve something similar to the pseudo-code below. We highlight in yellow effectful definitions and operations.

```
;; e1 ⇓E v1
(define* v1 (d:eval-exp* env e1))
;; E' ← E + [x := v1]
(define* env2 (environ-push* env y v1))
;; e2 ⇓E' v2
(define* v2 (d:eval-exp* env2 e2))
```

Roadmap: abstracting effectful computation

Combining:

- **Effectful operations:** `s:eval-exp` and `environ-push`, with
- **Effectful variable declaration:** `v1`, `env2`, and `v2`

```
;; e1 ⇓E v1
(define* v1 (d:eval-exp* env e1))
;; E' ← E + [x := v1]
(define* env2 (environ-push* env y v1))
;; e2 ⇓E' v2
(define* v2 (d:eval-exp* env2 e2))
```

$$e_1 \Downarrow_E v_1 \blacktriangleright E' \leftarrow E + [y := v] \blacktriangleright e_2 \Downarrow_{E'} v_2$$

A proxy example

Arithmetic on the heap

Example

> Consider a heap of integers. We allocate two integers and then a third integer that holds the some of the first two.

```
(define (prog1 h1)
  ;; allocate x with 1
  (define eff-x (heap-alloc h1 1))
  (define x (eff-result eff-x))
  (define h2 (eff-state eff-x))
  ;; allocate y with 2
  (define eff-y (heap-alloc h2 2))
  (define y (eff-result eff-y))
  (define h3 (eff-state eff-y))
  ;; allocate z with (+ x y)
  (heap-alloc h3 (+ (heap-get h3 y) (heap-get h3 x))))
(define (run-state h op) (eff-state (op h)))
(define H (heap (hash (handle 0) 1 (handle 1) 2 (handle 2) 3)))
(check-equal? (run-state empty-heap prog1) H)
```

Effectful operations

- An **effectful operation** takes a state and returns an effect `eff` that pairs some state with some result. An effectful operation is parameterized by the state type and by the result type.
- Below we define two effectful operations where the state is a heap.

Alloc

```
(define (num x)
  (lambda (h)
    (heap-alloc h x)))
```

Add

```
(define (add x y)
  (lambda (h)
    (heap-alloc h (+ (heap-get h y) (heap-get h x)))))
```

Did you know?

> - The state (heap) is a parameter, so that we can combine effectful operations.
> - Functions `num` and `add` each returns an effectful operation

Sequencing effectful operations

Example

```
(define (prog2 h1)
  ;; allocate x with 1
  (define eff-x ((num 1) h1))
  (define x (eff-result eff-x))
  (define h2 (eff-state eff-x))
  ;; allocate x with 2
  (define eff-y ((num 2) h2))
  (define y (eff-result eff-y))
  (define h3 (eff-state eff-y))
  ;; allocate y with (+ x y)
  ((add x y) h3))
```

The bind operator

```
(define (bind o1 o2)
  (lambda (h1)
    (define eff-r (o1 h1))
    (define r (eff-result eff-r))
    (define h2 (eff-state eff-r))
    ((o2 r) h2)))
```

> We highlight in yellow an example of redundant code. Function `bind` abstracts away the redundant code.

Abstracting with bind

Before

```
(define (prog2 h1)
  ;; allocate x with 1
  (define eff-x ((num 1) h1))
  (define x (eff-result eff-x))
  (define h2 (eff-state eff-x))
  ;; allocate x with 2
  (define eff-y ((num 2) h2))
  (define y (eff-result eff-y))
  (define h3 (eff-state eff-y))
  ;; allocate y with (+ x y)
  ((add x y) h3))
```

After

```
(define prog3
  ;; allocate x with 1
  (bind (num 1)
    (lambda (x)
      ;; allocate x with 2
      (bind (num 2)
        (lambda (y)
          ;; allocate y with (+ x y)
          (add x y))))))
```

> Using the bind operator we remove redundant code. You can think of bind as a variable declaration, akin to an effectful define.

Stack machines

The state does not need to be a heap

Stack machines

- Uses a stack of number to represent memory (rather than registers)
- Variable-free code
- Very compact object code
- Examples of (virtual) stack machines: OpenJDK JVM, CPython interpreter

```
def mult():
  x = pop
  y = pop
  push (x * y)
def prog():
  push(2)
  push(5)
  mult() # 2 * 5 = 10
  push(2)
  mult()  # 10 * 2 = 20
```

A stack-based evaluator

Operations

- `push(n) -> (void)`
- `pop() -> number`

State

A stack-based evaluator

Operations

- `push(n) -> (void)`
- `pop() -> number`

State

- a list of numbers

Implementing pop

```
(define (pop)
```

Implementing pop

```
(define (pop)

  (lambda (stack)
    (eff (rest stack) (first stack))))
```

Implementing push

```
(define (push n)
```

Implementing push

```
(define (push n)

  (lambda (stack)
    (eff (cons n stack) (void))))
```

Implementing `mult`

```
def mult():
  x = pop
  y = pop
  push (x * y)
```

Implementing `mult`

```python
def mult():
  x = pop
  y = pop
  push (x * y)
```

```scheme
(define (mult)
  (bind (pop)
    (lambda (x)
      (bind (pop)
        (lambda (y)
          (push (* x y)))))))
```

Implementing prog

Pseudo Code

```
def prog():
  push(2)
  push(5)
  mult() # 2 * 5 = 10
  push(2)
  mult()  # 10 * 2 = 20
```

Implementing prog

Pseudo Code

```
def prog():
  push(2)
  push(5)
  mult() # 2 * 5 = 10
  push(2)
  mult()  # 10 * 2 = 20
```

In Racket

```
(define prog4
  (bind (push 2)
    (lambda (x1)
      (bind (push 5)
        (lambda (x2)
          (bind (mult)
            (lambda (x3)
              (bind (push 2)
                (lambda (x4)
                  (mult)))))))))
```

```
(check-equal? (run-state (list) prog4) (list 20))
```

> Unfortunately, the code appears very nested if we indent it as we usually do. **Can we do better?**

Sequencing effectful operators

Sequencing effectful operators

Solution

```
(define (seq2 op1 op2)
  (bind op1 (lambda (x) op2)))

(define (seq op . ops)
  (cond [(empty? ops) op]
        [else (seq2 op (apply seq ops))]))
```

Revisit prog4

Sequencing effectful operators

Solution

```
(define (seq2 op1 op2)
  (bind op1 (lambda (x) op2)))

(define (seq op . ops)
  (cond [(empty? ops) op]
        [else (seq2 op (apply seq ops))]))
```

Revisit prog4

```
(define prog5
  (seq (push 2)
       (push 5)
       (mult)
       (push 2)
       (mult)))
(check-equal? (run-state (list) prog5)
  (list 20))
```

Limitations

> The seq operator is a regular function call, which takes ***expressions*** as its arguments. This complicates a situation where we might need to create a temporary variable (say to cache a result) in the middle of a seq.

Syntactic sugar

for stateful operations

Syntactic sugar: the do notation

> Macros can be useful technique to avoid redundant code. In our case, we are using a macro to avoid syntactic verbosity.

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

> You do not need to understand this code today. We will learn about macros in detail in a future lesson.

Syntactic sugar: the do notation

The do notation allows us to make our code less nested. The cost of using macros is that they obfuscate the program's semantics.

Before

```
(define (mult)
  (bind (pop)
    (lambda (x)
      (bind (pop)
        (lambda (y)
          (push (* x y)))))))
```

After

```
(define (mult)
  (do
    x ← (pop)
    y ← (pop)
    (push (* x y))))
```

Limitations

Similarly to `seq`, because of how the macro was designed, it takes a sequence of expressions. Monadic interfaces usually introduce an operator `pure` to workaround the issue.

The pure operator

The `pure` operator simply converts a pure (non-effectful) value into an effectful value, leaving the state unaltered. One useful benefit of this is that it allow us to combine effectful and pure operations in the same interface.

The pure operator

```
(define (pure v)
  (lambda (st)
    (eff st v)))
```

Example

```
(define (mult)
  (do
    x ← (pop)
    y ← (pop)
    z ← (pure (* x y))
    (push z)))
```

Summary: the monad

A monad is a **functional pattern** which can be categorized of two base combinators:

- **Bind:** combines two effectful operations o_1 and o_2. Operation o_1 produces a value that is consumed by operation o_2.
- **Pure:** Converts a pure value to a monadic operation, which can then be chained with `bind`.

> In this course, we will learn that the monadic pattern appears in different contexts.

Summary: the state monad

- **Data:** the monadic data is a pair (struct `eff`) that holds the global state and some result.
- **Bind:** combines operation o_1 with operation o_2; after executing o_1, we get a new state and some result that are both fed into operation o_2.

To think...

> Monadic function application: can we create a function call where all arguments are monadic values? What about a monadic map? And a monadic fold?

```
(define (mult)
  (do
    z ← (mapply * (pop) (pop))
    (push z)))
  ;; Or, simply: (define (mult) (bind (mapply * (pop) (pop)) push))
```