CS450

Structure of Higher Level Languages

Lecture 28: Effectful operations

Tiago Cogumbreiro

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

UMASS BOSTON

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`

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

```
(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: 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 Lecture 19.

Syntactic sugar: the do notation

UMASS BOSTON

> 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))
```