Chapter 10 :: Functional Languages

Evaluation Order

Michael L. Scott

ELSEVIER

Evaluation Order Revisited

- Applicative order: evaluates all arguments before invoking function
  - what you're used to in imperative languages
  - usually faster
- Normal order: doesn’t evaluate arg until you need it
  - sometimes faster
  - terminates if anything will (Church-Rosser theorem)

Evaluation Order (Example)

```
(and (not (= y 0)) (/ x y))
```

Why normal order may be slow

```
(define double (lambda (x) (+ x x )))
(double (* 3 4))
```

Applicative Order

(double (* 3 4))
➔ (double 12)
➔ (+ 12 12)
➔ 24

Normal Order

(double (* 3 4))
➔ (+ (* 3 4) (* 3 4))
➔ (+ 12 (* 3 4))
➔ (+ 12 12)
➔ 24

Scheme Evaluation Order

- In Scheme
  - functions use applicative order defined with lambda
    - arguments are evaluated right to left
  - special forms (aka macros) use normal order defined with syntax-rules

Scheme Applicative Order Example

```
(define add (lambda (x) (+ x 20)))
(define min (lambda (x y) (if (< x y) x y)))
(trace add)
(min (add 5) (add 20))
[Entering #[compound-procedure 4 add] Args: 20]
[40
        <== #[compound-procedure 4 add]  Args: 20]  ; <==
        means exiting this fct
[Entering #[compound-procedure 4 add] Args: 5]
 [25
         <== #[compound-procedure 4 add] Args: 5]
;Value: 25
```

Strict versus Non-strict Languages

- A *strict* language requires all arguments to be well-defined, so applicative order can be used
- A n*on-strict* language does not require all arguments to be well-defined; it requires normal-order evaluation
- Scheme is strict for functions, but non-strict for special forms
- C is strict, except for boolean expressions

Forcing Normal Order in Scheme

- Use **delay** and **force** constructs
  - delay: creates an expression but does not evaluate it
  - force: forces the evaluation of a delayed expression
- Example

  (define expr (delay (+ a 10)))

  (define a 15)

  (force expr) ➔ 25

Creating Generator Functions in Scheme

naturals is an infinite list of natural numbers, but we can't actually generate an infinite list so we generate one natural number at a time and then delay the next call to the natural number generator function until we are ready for the next number

```
(define naturals
    (letrec ((next (lambda (n)
        ; the delay prevents an infinite recursion
         (cons n (delay (next (+ n 1)))))))
      (next 1)))
; tail forces the next natural number to be generated
(define tail (lambda (stream) (force (cdr stream))))
```

(car naturals) ➔ 1

(car (tail naturals)) ➔ 2

(car (tail (tail naturals))) ➔ 3

How to Create and Use a Generic Generator Function

1. Create: Define a “list” where you cons the next item in the stream with a delayed call to the generator function.
   a. You may need to perform a computation using the current parameters to the generator function to obtain the next item
   b. The delayed call should contain the parameters to create the subsequent item
2. Use: Create a function that takes your “list” as an argument
   a. Print the next item by taking the car of the list
   b. Make a recursive call to your function and pass it the cdr of your list, prefixed with the force function to force the next evaluation of the generator function (tail performs this task on the previous slide)

Memoization

- Memoization: Technique saves an expression's result in some type of fast lookup structure
  - Thereafter references to the expression use this computed value
  - Brings performance of normal order evaluation within a constant factor of applicative order evaluation
- Spreadsheets use memoization

Example:

a10 = b10 + c10

b10 = 3 * b9

c10 = 8 * c9

b9 = 5

c9 = 10

Memoization (Potential Problem)

- May not work properly in the presence of side-effects
- Example:

```
(define x 5)
(define y 10)
(define (z (* x y))
(set! x 2)
(define (a (* x y))
```