Functional Programming

Lazy Evaluation

H. Turgut Uyar

2013-2016

License

Topics

1. Expression Evaluation
   - Strategies
   - Short-Circuit Evaluation
   - Space Complexity
2. Infinite Lists
   - Introduction
   - Generators
   - Folding

Expression Evaluation

- reduce an expression to a value
- substitution model

- take operator with lowest precedence
- evaluate its operands (note the recursion)
- apply operator to operands
- substitute expression with value

- evaluating a name: substitute it with its definition

Expression Evaluation Example

```
(3.14159 * r) * r
-- r = 2.4
(3.14159 * 2.4) * r
7.539815999999999 * r
7.539815999999999 * 2.4
18.095558399999998
```


Function Evaluation

- evaluate all actual parameters (left to right)
- substitute function application with its definition
- substitute formal parameters with actual parameters

Function Evaluation Example

```
sqr :: Integer -> Integer
sqr x = x * x

sumOfSquares :: Integer -> Integer -> Integer
sumOfSquares x y = sqr x + sqr y

-- x = 3, y = 2 + 2
```


Function Evaluation Example

```
sumOfSquares 3 (2 + 2)
sumOfSquares 3 4
sqr 3 + sqr 4
(3 * 3) + sqr 4
9 + sqr 4
9 + (4 * 4)
9 + 16
25
```

Evaluation Strategies

- strict: evaluate parameters, apply function ("call by value")
- normal order: evaluate parameters when needed ("call by name")

Church-Rosser property

result is the same as long as:

- there are no side effects
- all evaluations terminate

Normal Order Evaluation Example

```
sumOfSquares 3 (2 + 2)
sqr 3 + sqr (2 + 2)
(3 * 3) + sqr (2 + 2)
9 + sqr (2 + 2)
9 + (2 + 2) * (2 + 2)
9 + 4 * (2 + 2)
9 + 4 * 4
9 + 16
25
```


Lazy Evaluation

- strict evaluation evaluates parameters only once
- but it might evaluate parameters which are not needed
- normal order evaluation doesn't evaluate parameters which are not needed
- but it might evaluate others more than once
- lazy evaluation: evaluate parameter once when *first* needed
- memoization

Lazy Evaluation Example

```
sumOfSquares 3 (2 + 2)
sqr 3 + sqr (2 + 2)
(3 * 3) + sqr (2 + 2)
9 + sqr (2 + 2)
9 + (2 + 2) * (2 + 2)
9 + 4 * 4
9 + 16
25
```

Evaluation Strategies

- most languages use strict evaluation

Python

```
def first(x, y):
    return x

# first(1, 1 // 0) ~> division by zero
```


Evaluation Strategies

- Haskell uses lazy evaluation by default

Haskell

```
first :: Integer -> Integer -> Integer
first x y = x

-- first 1 (1 ‘div‘ 0) ~> 1
```


Short-Circuit Evaluation

- short-circuit evaluation:
  evaluation stops as soon as result is determined

C

```
(a >= 0) && (b < 10)
// second clause not evaluated if a < 0

(a >= 0) || (b < 10)
// second clause not evaluated if a >= 0

(a >= 0) || (b++ < 10)
// dangerous
```


Short-Circuit Evaluation

- code might depend on short-circuit evaluation

Java

```
// find the index of a key item in a list
index = 0;
while ((index < items.length) && (items[index] != key))
    index++;
```

Short-Circuit Evaluation Examples

```
and :: Bool -> Bool -> Bool
and x y = if x then y else False

or :: Bool -> Bool -> Bool
or x y = if x then True else y
```


Space Complexity

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n - 1)
```

- not tail-recursive
- creates new stack frames

Space Complexity Example

```
facIter :: Integer -> Integer -> Integer
facIter acc 0 = acc
facIter acc n = facIter (acc * n) (n - 1)
```

- tail-recursive
- lazy evaluation: doesn't multiply until the last moment

Space Complexity Example

```
facIter 1 n
~> facIter (1*n) (n-1)
~> facIter ((1*n)*(n-1)) (n-2)
~> facIter (((1*n)*(n-1))*(n-2)) (n-3)
~> ...
```

Space Complexity

- possible solution: make the value needed

```
facIter acc 0 = acc
facIter acc n
  | acc == acc = facIter (acc * n) (n - 1)
```


Strictness

- force the evaluation of a parameter: `seq`

```
seq :: a -> b -> b
seq x y
  | x == x = y  -- evaluate x and return y

facIter :: Integer -> Integer -> Integer
facIter acc 0 = acc
facIter acc n = seq acc (facIter (acc * n) (n - 1))
```


Strictness

- make a function strict on a parameter: `strict`

```
strict :: (a -> b) -> a -> b
strict f x = seq x (f x)

fac :: Integer -> Integer
fac n = facIter 1 n
  where
    facIter :: Integer -> Integer -> Integer
    facIter acc 0  = acc
    facIter acc n' = strict facIter (acc * n') (n' - 1)
```


Infinite Lists

- lazy evaluation makes it possible to work with infinite data structures
- create a list with infinite copies of the same element: `repeat 42 ~> [42, 42, 42, ...]`

```
repeat :: a -> [a]
repeat x = x : repeat x

addFirstTwo :: Num a => [a] -> a
addFirstTwo (x1:x2:_) = x1 + x2

-- addFirstTwo (repeat 42) ~> 84
```

Infinite Ranges

```
from :: Integer -> [Integer]
from n = n : from (n + 1)

-- from 5 -> [5, 6, 7, 8, ...]

-- OR: [5 ..]
-- addFirstTwo [7 ..] ~> 15
```


Infinite List Example

Fibonacci sequence

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

-- take 5 fibs ~> [1, 1, 2, 3, 5]
```


Infinite List Example

sieve of Eratosthenes

```
sieve :: [Integer] -> [Integer]
sieve (x:xs) = x : sieve [y | y <- xs, y ‘mod‘ x > 0]

-- sieve [2 ..] -> [2, 3, 5, 7, 11, ...]

primes :: [Integer]
primes = sieve [2 ..]
```


Infinite List Example

prime number test

```
isPrime :: Integer -> Bool
isPrime n = elemOrd n primes

elemOrd :: Ord a => a -> [a] -> Bool
elemOrd n (x:xs)
  | n <  x    = False
  | n == x    = True
  | otherwise = elemOrd n xs
```

Generators

- Python uses generators for computing values when needed
- `yield` instead of `return`

example

```
def repeat(n):
    while True:
        yield n


answers = repeat(42)
for x in answers:
    print(x)    # 42, 42, 42, ...
```


Generators

- next call continues from where previous call left off

example

```
def from_(n):
    while True:
        yield n
        n += 1


from5 = from_(5)
for x in from5:
    print(x)    # 5, 6, 7, ...
```


Generator Example

Fibonacci sequence

```
def fibs():
    yield 1
    yield 1
    back1, back2 = 1, 1
    while True:
        num = back2 + back1
        yield num
        back1 = back2
        back2 = num
```


Folding

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f s []     = s
foldl f s (x:xs) = foldl f (f s x) xs
```

- `foldr`: not tail recursive
- `foldl`: tail recursive

Folding Example

```
foldl (*) 1 [1 .. n]
~> foldl (*) (1*1) [2 .. n]
~> foldl (*) ((1*1)*2) [3 .. n]
~> foldl (*) (((1*1)*2)*4) [4 .. n]
~> ...
```


Folding

- `foldl'`: strict `foldl`

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f s []     = s
foldl' f s (x:xs) = strict (foldl' f) (f s x) xs
```


Folding Example

```
foldl' (*) 1 [1 .. n]
~> foldl' (*) 1 [2 .. n]
~> foldl' (*) 2 [3 .. n]
~> foldl' (*) 6 [4 .. n]
~> ...
```


Folding Example

```
foldl (&&) True (repeat False)
~> foldl (&&) True [False, False, False, ...]
~> foldl (&&) ((&&) True False) [False, False, ...]
~> foldl (&&) False [False, False, ...]
~> foldl (&&) ((&&) False False) [False, ...]
~> foldl (&&) False [False, ...]
~> ...
-- never terminates
```

Folding Example

```
foldr (&&) True (repeat False)
~> foldr (&&) True [False, False, False, ...]
~> (&&) False (foldr (&&) True [False, False, ...])
~> False
```


Folding Example

```
foldr ((:).(+2)) [] [1 .. n]
~> ((:).(+2)) 1 (foldr ((:).(+2)) [] [2 .. n])
~> (1+2) : (foldr ((:).(+2)) [] [2 .. n])
~> 3 : (foldr ((:).(+2)) [] [2 .. n])
~> 3 : ((:).(+2)) 2 (foldr ((:).(+2)) [] [3 .. n])
~> 3 : 4 : (foldr ((:).(+2)) [] [3 .. n])
~> ...
```

- space complexity: $O(1)$

Folding Example

```
foldr (*) 1 [1 .. n]
~> (*) 1 (foldr (*) 1 [2 .. n])
~> (*) 1 ((*) 2 (foldr (*) 1 [3 .. n]))
~> (*) 1 ((*) 2 ((*) 3 (foldr (*) 1 [4 .. n])))
~> ...
```

- space complexity: $O(n)$

Folding Strategies

- if f is lazy on its second argument, prefer `foldr`
- if the whole list will be traversed, prefer `foldl'`

References

Required Reading: Thompson

- Chapter 17: Lazy programming
- Chapter 20: Time and space behaviour
  - 20.5: Folding revisited