Functional Programming

Function Closures

H. Turgut Uyar

2013-2016

License

Topics

1. Function Closures
   - Functions as Result
   - Decorators
   - Currying
2. Function Operators
   - Composition
   - Application
   - Operator Sections

Functions as Result

- higher-order functions can return functions as result

example: body surface area

- h: height (cm), w: weight (kg), result: area (m^2)
- Du Bois formula: $0.007184 \cdot h^{0.725} \cdot w^{0.425}$
- Boyd formula: $0.0333 \cdot h^{0.3} \cdot w^{0.6157 - 0.0188 \log_{10} w}$
- Boyd formula more accurate in infants

Returning Function Example

```
duBois :: Float -> Float -> Float
duBois h w = 0.007184 * (h ** 0.725) * (w ** 0.425)

boyd :: Float -> Float -> Float
boyd h w = 0.0333 * (h ** 0.3)
         * (w ** (0.6157 - 0.0188 * (logBase 10 w)))

bsa :: Integer -> (Float -> Float -> Float)
bsa age = if age < 3 then boyd else duBois

-- (bsa 20) 180 75 ~> 1.9424062
-- (bsa 2)  86 13  ~> 0.58253276
```

Function Closure

- function value has two parts:
- code
- environment current at the time of definition
- function closure

Function Closure Example

```
stepRange :: Integer ->
                (Integer -> Integer -> [Integer])
stepRange step = getRange
  where
    getRange :: Integer -> Integer -> [Integer]
    getRange m n = [m, m + step .. n]

step1 :: Integer -> Integer -> [Integer]
step1 = stepRange 1
-- step1 3 7 ~> [3, 4, 5, 6, 7]

step5 :: Integer -> Integer -> [Integer]
step5 = stepRange 5
-- step5 9 20 ~> [9, 14, 19]
```

Function Closure Example

Python

```
def step_range(step):
    def get_range(m, n):
        return range(m, n + 1, step)
    return get_range

step1 = step_range(1)

# step1(3, 7) ~> [3, 4, 5, 6, 7]
```

Decorators (Python)

- decorator: takes function as parameter, returns transformed function
- `@decorator`

example: entry and exit messages

```
def entry_exit(f):
    def wrapped(x):
        print("Entering with parameter: %s" % x)
        result = f(x)
        print("Exiting with result: %s" % result)
        return result
    return wrapped
```

Decorator Example

```
def fac(n):
    return 1 if n == 0 else n * fac(n - 1)

# entry_exit(fac)(5)


@entry_exit
def fac(n):
    return 1 if n == 0 else n * fac(n - 1)

# fac(5)
```

Decorator Example

memoization

```
def memoize(f):
    cache = {}
    def wrapped(x):
        if x not in cache:
            cache[x] = f(x)
        return cache[x]
    return wrapped
```

Decorator Example

memoized Fibonacci sequence

```
@memoize
def fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

Currying

- function with two input parameters:
  function with one input parameter,
  returns function with one input parameter
- generalize for *n* parameters: currying
- partial application: call with fewer paratemers,
  obtain function that expects remaining parameters
- in function signatures, arrows associate to the right
- function application associates to the left

Currying Example

```
add :: Integer -> Integer -> Integer
add x y = x + y

-- same as:
add :: Integer -> (Integer -> Integer)
add x = \y -> x + y

increment :: Integer -> Integer
increment = add 1
-- increment = \y -> 1 + y
-- increment y = 1 + y
```

Currying Examples

```
floorAll xs = map floor xs

-- same as:
floorAll = map floor

allOdds xs = filter odd xs

-- same as:
allOdds = filter odd
```

Currying Example

```
stepRange :: Integer -> Integer -> Integer ->
                 [Integer]
stepRange step m n = [m, m + step .. n]

step1 :: Integer -> Integer -> [Integer]
-- step1 m n = stepRange 1 m n
step1 = stepRange 1

naturals :: Integer -> [Integer]
-- naturals n = stepRange 1 0 n
naturals = stepRange 1 0
-- naturals n = step1 0 n
-- naturals = step1 0
```

Currying Functions

- curry: convert a function that takes a pair
  into an equivalent function that takes two parameters

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f = \x y -> f (x, y)

-- same as:
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

Curry Example

```
addT :: (Integer, Integer) -> Integer
addT (x, y) = x + y

addC = curry addT
```

- exercise: convert a function that takes two parameters
  into an equivalent function that takes a pair:
  `uncurry addC ~> addT`

Function Composition

function composition

```
(f . g) x = f (g x)
```

- what is the type of (.)?

```
(.) : (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
infixr 9 .
```

Function Composition Examples

test whether number is even

```
even :: Integer -> Bool
even = not . odd
```

second element of a list

```
second :: [a] -> a
second = head . tail
```

Function Composition Examples

last element of a list

```
last :: [a] -> a
last = head . reverse
```

length of a list

```
length :: [a] -> Int
length = sum . map (\_ -> 1)
```

Function Application

function application

```
f $ x = f x
```

- what is the type of `($)`?

```
($) :: (a -> b) -> a -> b
f $ x = f x
infixr 0 $
```

- why?

Function Application

- less parentheses, more readable

```
sum (filter odd (map (floor . sqrt) [1 .. 100]))

-- same as:
sum $ filter odd $ map (floor . sqrt) [1 .. 100]
```

Function Application

- needed in some cases

```
zipWith ($) [sum, product] [[1, 2], [3, 4]]
-- [3, 12]
```

Operator Sections

- operators can be partially applied
- a function that expects the missing argument

```
(+2) 5  ~> 7
(>2) 5  ~> True
(2>) 5  ~> False

filter (4>)      [5, 2, 3, 7]  ~> [2, 3]
map    (‘div‘ 2) [5, 2, 3, 7]  ~> [2, 1, 1, 3]

(map (*2) . filter ((==1) . (‘mod‘ 2))) [5, 2, 3, 6]
```

References

Required Reading: Thompson

- Chapter 11: Higher-order functions