Function Haskell Exercises

Young W. Lim

2018-10-08 Mon

Outline

Based on

"The Haskell Road to Logic, Maths, and Programming",
K. Doets and J. V. Eijck

Using FCT.hs

```
module FCT

where

import List
```

```
:load FCT
```

Relation Composition

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load FCT
[1 of 1] Compiling FCT              ( FCT.hs, interpreted )
Ok, modules loaded: FCT.
*FCT> image (*2) [1,2,3]
[2,4,6]
*FCT> injective (*2) [1, 2, 3]
True
*FCT> injective (^2) [1, 2, 3]
True
*FCT> injective (^2) [-1, -2, -3]
True
*FCT> injective (*2) [-1, -2, -3]
True
*FCT> surjective (*2) [1, 2, 3] [0, 1, 2, 3]
False
*FCT> surjective (*2) [1, 2, 3] [1, 2, 3]
False
*FCT> surjective (*2) [1, 2, 3] [2, 4, 6]
True
```

Basic Notations

- $f(x) = x^2 + 1$

```
f x = x^2 + 1
```

- $abs(x) = |x|$

```
absReal x | x >= 0    = x
          | otherwise = -x |
```

- the identity function

```
id :: a -> a
ix x = x
```

From a function to a list

- converting a function to a list

```
list2fct :: Eq a => [(a,b)] -> a -> b
list2fct [] _ = error "function not total"
list2fct ((u,v):uvs) x | x == u    = v
                       | otherwise = list2fct uvs x
```

- examples

```
*Main> fct2list f [1, 2, 3]
[(1,2),(2,5),(3,10)]
*Main> lst = fct2list f [1, 2, 3]
*Main> lst
[(1,2),(2,5),(3,10)]
```

From a list to a function

- converting a list to a function

```
fct2list :: (a -> b) -> [a] -> [(a,b)]
fct2list f xs = [ (x, f x) | x <- xs ]
```

- examples

```
*Main> lst
[(1,2),(2,5),(3,10)]
*Main> list2fct lst 1
2
*Main> list2fct lst 2
5
*Main> list2fct lst 3
10
```

Range of a function (1)

- when a function is implemented as a list of pairs

```
ranPairs :: Eq b => [(a,b)] -> [b]
ranPairs f = nub [ y | (_,y) <- f ]
```

- examples

```
*Main> ranPairs [(1,3), (2,3), (3,4), (4,3), (5,4)]
[3,4]
```

Range of a function (2)

- when a function is defined on an enumerable domain

```
listValues  :: Enum a => (a -> b) -> a -> [b]
listValues f i = (f i) : listValues f (succ i)
```

- examples

```
*Main> f x = x^2 + x + 1
*Main> listValues f 1
[3,7,13,21,31,43,57,73,91,111,133,157,183,211,241,273,307,343,381,421,
 463,507,553,601,651,703,757,813,871,931,993,1057,1123,1191,1261,1333,
 1407,1483,1561,1641,1723,1807,1893,1981,2071,2163,2257,2353,2451,2551,
 2653,2757,2863,2971,3081,3193,^CInterrupted
```

Range of a function (3)

- when a function has a bounded domain

```
listRange :: (Bounded a, Enum a) => (a -> b) -> [b]
listRange f = [ f i | i <- [minBound..maxBound] ]
```

- examples

```
f x | x == A = 1
    | x == B = 2
    | x == C = 3

data X = A | B | C deriving (Bounded, Enum, Eq, Show)
next A = B
next B = C

*Main> maxBound :: X
C
*Main> minBound :: X
A
*Main> listRange f
[1,2,3]
```

nub function

- `nub : :  Eq a => [a] -> [a]`
- the nub function removes duplicate elements from a list.
- in particular, it keeps only the first occurrence of each element.
- (The name nub means ‘essence’.)

Enum Class

- Enum Class has instances of Char, Int, Integer, Float, Double

```
class  Enum a  where
    succ, pred        :: a -> a
    toEnum            :: Int -> a
    fromEnum          :: a -> Int
    enumFrom          :: a -> [a]                -- [n..]
    enumFromThen      :: a -> a -> [a]           -- [n,n'..]
    enumFromTo        :: a -> a -> [a]           -- [n..m]
    enumFromThenTo    :: a -> a -> a -> [a]      -- [n,n'..m]
```

Enumeration

- `[1..5]` : a list of numbers `[1,2,3,4,5]`
- `[10..]` : an infinite list of numbers `[10, 11, 12, ...]`
- `[5..1]` : an empty list
- `[5,4,3,2,1]` : not empty
- `[0, -1 ..]` : a list of negative integers
- `[-5..-1]` : syntax errror, must be `[-5.. -1]` (space)
- `[1,3..9]` : equal to `[1,3,5,7,9]`
- `[-1,3,..9]` : equal to `[-1,3,7]`

Bounded Class

- Bounded Class has instances of Char, Int
- the Bounded class is used to name the upper and lower limits of a type.
- Ord is not a superclass of Bounded since types that are not totally ordered may also have upper a

```
class  Bounded a  where
 minBound         :: a
 maxBound         :: a

instance Bounded Char where
   minBound            =  '\0'
   maxBound            =  '\xffff'
```

More than one agrument functions

- three argument function type

  `f :: (a, b, c) -> d`
- series of one argument function type

  `f :: a -> b -> c -> d`

```
 f x  y z           f :: a ->  b -> c -> d
(f x) y z           f :: a -> (b -> c -> d)
 g y  z             g :: b ->  c -> d
(g y) z             g :: b -> (c -> d)
 h z                h :: c -> d
```

  - f x ⇒ g
  - g y ⇒ h
  - h z ⇒ f x y z

curry and uncurry

- curry operation

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

- uncurry operation

```
uncurry :: (a->b->c) -> ((a,b)->c)
uncurry f (a,b)= f a b
```

- examples

```
f2 x y = x + y        -- curried form
g2 (x,y) = x + y      -- uncurried form

*Main> f2 3 4
7
*Main> g2 (3,4)
7
*Main> uncurry f2 (3,4)
7
*Main> curry g2 (3,4)
7
```

curry3 and uncurry3

- curry3 operation

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x,y,z)
```

- uncurry3 operation

```
uncurry3 :: (a -> b -> c -> d) -> (a,b,c) -> d
uncurry3 f (x,y,z) = f x y z
```

- examples

```
f3 x y z = x + y + z      -- curried form
g3 (x,y,z) = x + y + z    -- uncurried form

*Main> f3 3 4 5
12
*Main> g3 (3,4,5)
12
*Main> uncurry f3 (3,4,5)
12
*Main> curry g3 (3,4,5)
12
```

Closed form function examples

- `f1 x = x^2 + 2*x + 1`
- `f1' = \x -> x^2 + 2*x + 1`
- `g1 x = (x+1)^2`
- `g1' = \x -> (x+1)^2`
- examples

```
*Main> fmap f1 [1..10]
[4,9,16,25,36,49,64,81,100,121]
*Main> fmap f1' [1..10]
[4,9,16,25,36,49,64,81,100,121]
*Main> fmap g1' [1..10]
[4,9,16,25,36,49,64,81,100,121]
*Main> fmap g1 [1..10]
[4,9,16,25,36,49,64,81,100,121]
```

Recurrence form function examples

- ```
  g 0 = 0
  g n = g (n-1) + n
  ```
- ```
  h 0 = 0
  h n = h (n-1) + (2*n)
  ```
- ```
  k 0 = 0
  k n = k (n-1) + (2*n-1)
  ```
- ```
  fac 0 = 1
  fac n = fac (n-1) * n
  ```
- examples
  ```
  *Main> g 3
  6
  *Main> h 3
  12
  *Main> k 3
  9
  *Main> fac 3
  6
  ```

Recurrence vs closed forms

- recurrence (recursion)

```
g 0 = 0
g n = g (n-1) + n

g 3 = g 2 + 3...............
......g 2 = g 1 + 2.........
............g 1 = g 0 + 1....
..................g 0 = 0....
............g 1 = 0 + 1 = 1..
......g 2 = 1 + 2 = 3.......
g 3 = 3 + 3 = 6.............
```

- closed form

```
g' n = ((n + 1) * n ) / 2

g' 3 = ((3+1)*3) / 2
```

Recurrence vs iteration forms

- recurrence (recursion)

```
fac 0 = 1
fac n = fac (n-1) * n

(fac 3) = (fac 2) * 3.......................
..........(fac 2) = (fac 1) * 2.............
....................(fac 1) = (fac 0) * 1....
..............................(fac 0) = 1....
....................(fac 1) = 1 * 1 = 1......
..........(fac 2) = 1 * 2 = 2...............
(fac 3) = 2 * 3 = 6.........................
```

- iteration

```
fac' n = product [1..n]

fac' 3 = product [1..3]
       = product [1, 2, 3]
       = 1 * 2 * 3 = 6
```

Restricting the domain of a function

- restricting the domain

```
restrict :: Eq a => (a -> b) -> [a] -> a -> b
restrict f xs x | elem x xs = f x
                | otherwise = error "argument not in domain"
```

- example

```
f x = x^2 + 1

*Main> restrict f [-4..4] (-2)
5

*Main> restrict f [-4..4] (-20)
*** Exception: argument not in domain
CallStack (from HasCallStack):
  error, called at func1.hs:63:31 in main:Main

*Main> restrict f [-4..4] -20  --> parenthesis is required
```

Restricting the domain of a pair-wise function

- restricting the domain

```
restrictPairs :: Eq a => [(a,b)] -> [a] -> [(a,b)]
restrictPairs xys xs = [ (x,y) |  (x,y) <- xys, elem x xs ]
```

- example

```
*Main> restrictPairs [(1,3), (2,3), (4,5), (5,7), (7,9)] [1..4]
[(1,3),(2,3),(4,5)]
```

Image and CoImage

- image and coImage

```
image :: Eq b => (a -> b) -> [a] -> [b]
image f xs = nub [ f x | x <- xs ]

coImage :: Eq b => (a -> b) -> [a] -> [b] -> [a]
coImage f xs ys = [ x | x <- xs, elem (f x) ys ]
```

- example

```
*Main> image (*2) [1, 2, 3, 4, 5]
[2,4,6,8,10]
*Main> coImage (*2) [1, 2, 3, 4, 5] [2, 4, 6, 8]
[1,2,3,4]
*Main>
```

Image and CoImage Pairs

- image and coImage Pairs

```
imagePairs :: (Eq a, Eq b) => [(a,b)] -> [a] -> [b]
imagePairs f xs = nub [ y | (x,y) <- f, elem x xs]

coImagePairs :: (Eq a, Eq b) => [(a,b)] -> [b] -> [a]
coImagePairs f ys = [ x | (x,y) <- f, elem y ys]
```

- example

```
*Main> imagePairs [(1,2), (2,4), (3,6), (4,8), (5,10)] [1,2,3]
[2,4,6]
*Main> coImagePairs [(1,2) (2,4), (3,6), (4,8), (5,10)] [2,4,6,8]
[1,2,3,4]

*Main>
```

Injective and surjective functions

- image and coImage Pairs

```
injective :: Eq b => (a -> b) -> [a] ->  Bool
injective f [] = True
injective f (x:xs) =
   notElem (f x) (image f xs) && injective f xs

surjective :: Eq b => (a -> b) -> [a] -> [b] -> Bool
surjective f xs [] = True
surjective f xs (y:ys) =
   elem y (image f xs) && surjective f xs ys
```

- example

```
*Main> injective f [0..2]
True
*Main> injective f [-2..2]
False
*Main> surjective f [0, 1, 2] [1, 2, 5]
True
*Main> surjective f [0, 1, 2] [1, 2, 5, 6]
False
```

Inverse functions

- $f(x) = \frac{9}{5}x + 32$
- $f^{-1}(x) = \frac{5}{9}(x - 32)$
- Celcius to Fahrenheit, Fahrenheit to Celcius

```
c2f, f2c :: Int -> Int
c2f x = div (9 * x) 5 + 32
f2c x = div (5 * (x - 32)) 9
```

- example

```
*Main> c2f 26
78
*Main> f2c 78
25
```

Enum Class

- ```
  class Enum a where
        succ, pred :: a -> a
        toEnum :: Int -> a
        fromEnum :: a -> Int
  ```
- `fromEnum` should be left_inverse of `toEnum`
  `fromEnum (toEnum x) = x`
  ```

Enum Class Examples

- ```
  data MyDataType = Foo | Bar | Baz

  instance Enum MyDataType
    toEnum 0 = Foo
    toEnum 1 = Bar
    toEnum 2 = Baz

    fromEnum Foo = 0
    fromEnum Bar = 1
    fromEnum Baz = 2
  ```
- ```
  data MyDataType = Foo | Bar | Baz deriving (Enum)
  ```
- ```
  instance Enum MyDataType where
    fromEnum = fromJust . flip lookup table
    toEnum = fromJust . flip lookup (map swap table)
  table = [(Foo, 0), (Bar, 1), (Baz, 2)]
  ```

https://stackoverflow.com/questions/6000511/better-way-to-define-an-enum-in-haskell
  ```

ord and chr functions

- the `ord` and `chr` functions are `fromEnum` and `toEnum` restricted to the type Char
- ```
  ord :: Char -> Int
  ord = fromEnum

  ord 'a' ........ 97
  ord '\n' ....... 10
  ord 'NULL' .....  0
  ```
- ```
  chr :: Int -> Char
  chr = toEnum

  char 97 ........ 'a'
  char 10 ........ '\n'
  char 0  ........
  ```

http://zvon.org/other/haskell/Outputchar/chr_f.html
http://zvon.org/other/haskell/Outputchar/ord_f.html

successor functions

- ```
  succ0 :: Integer -> Integer
  succ0 x = x + 1
  ```
- ```
  succ1 :: Integer -> Integer
  succ1 = \ x -> if x < 0
                  then error "argument out of range"
                  else x+1
  ```
- ```
  succ2 :: Integer -> [Integer]
  succ2 = \ x -> if x < 0 then [] else [x+1]
  ```
- ```
  succ3 :: Integer -> Maybe Integer
  succ3 = \ x -> if x < 0 then Nothing else Just (x+1)
  ```
- ```
  *Main> fmap succ0 [1, 3, 5, 8]
  [2,4,6,9]
  *Main> fmap succ1 [1, 3, 5, 8]
  [2,4,6,9]
  *Main> fmap succ2 [1, 3, 5, 8]
  [[2],[4],[6],[9]]
  *Main> fmap succ3 [1, 3, 5, 8]
  [Just 2,Just 4,Just 6,Just 9]
  ```
  ```
  *Main> fmap succ0 [-1, 1, 3, 5, 7]
  [0,2,4,6,8]
  *Main> fmap succ1 [-1, 1, 3, 5, 7]
  [*** Exception: argument out of range
  CallStack (from HasCallStack):
    error, called at func1.hs:106:24 in main:M
  *Main> fmap succ2 [-1, 1, 3, 5, 7]
  [[],[2],[4],[6],[8]]
  *Main> fmap succ3 [-1, 1, 3, 5, 7]
  [Nothing,Just 2,Just 4,Just 6,Just 8]
  ```
  ```

partial functions

- a partial function from X to Y
  (f: X $\hookrightarrow$ Y) is a function
  f: X' $\rightarrow$ Y, for some proper subset X' of X.
- it generalizes the concept of a function f: X $\rightarrow$ Y
  by not forcing f to map every element of X
  to an element of Y (only some proper subset X' of X)
- if X' = X, then f is called a total function
  and is equivalent to a function.
- Partial functions are often used
  when the exact domain, X, is not known
  (e.g. many functions in computability theory)

composition of partial functions (1)

- ```
  pcomp :: (b -> [c]) -> (a -> [b]) -> a -> [c]
  pcomp g f = \ x -> concat [ g y | y <- f x ]
  ```
- Examples

  ```
  fn1 x = [x, x+1]
  gn1 x = [x^2, (x+1)^2]


  *Main> pcomp gn1 fn1 3
  [9,16,16,25]
  ```
  ```

composition of partial functions (2)

- ```
  mcomp :: (b -> Maybe c) -> (a -> Maybe b) -> a -> Maybe c
  mcomp g f = (maybe Nothing g) . f
  ```
- ```
  fn2 x
    | x < 0  = Nothing
    | x < 10 = Just x
    | otherwise = Nothing

  gn2 x
    | x < 0  = Nothing
    | x < 5 = Just (2*x)
    | otherwise = Nothing

  *Main> fmap (mcomp gn2 fn2) [-2..12]
  [Nothing,Nothing,Just 0,Just 2,Just 4,Just 6,Just 8,Nothing,
   Nothing,Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]
  *Main>
  ```

maybe method

- `maybe :: b -> (a -> b) -> Maybe a -> b`
- The maybe function takes
  - a default value `b`
  - a function `(a -> b)`
  - a Maybe value `Maybe a`
- If the Maybe value is Nothing,
  the function returns the default value
- Otherwise, it applies the function
  to the value inside the Just
  and returns the result.

Dealing with exceptions (1)

- converting `(a -> Maybe b)` functions into `(a -> b)` functions

```
part2error :: (a -> Maybe b) -> a -> b
part2error f = (maybe (error "value undefined") id) . f

f :: (a -> Maybe b)
part2error f :: a -> b

part2error f x = ((maybe (error "value undefined") id) . f) x
x :: a
f x :: Maybe b
part2error f x :: b

f :: (a -> Maybe b)
(maybe (error "value undefined") id) . f :: a -> b
(maybe (error "value undefined") id) :: Maybe b -> b
```

- `- maybe :: b -> (a -> b) -> Maybe a -> b`

Dealing with exceptions (2)

- ```
  maybe :: b -> (a -> b) -> Maybe a -> b
  maybe :: b -> (b -> b) -> Maybe b -> b

  (maybe (error "value undefined") id) :: Maybe b -> b

  (error "value undefined") :: b
  id :: b -> b

  y :: Maybe b   .... (f x :: Maybe b)
  (maybe (error "value undefined") id) y :: b
  ```
  ```

Dealing with exceptions (3)

- ```
  maybe :: b -> (b -> b) -> Maybe b -> b

  (error "value undefined") :: b
  id :: b -> b

  error :: [Char] -> a
  error :: [Char] -> b
  ```
- If the `Maybe b` type value is Nothing,
  the `maybe` function returns the default value
  `(error "value undefined")`
- Otherwise, it applies the `id` function
  to the `b` type value inside the Just
  and returns the `b` type result.
  ```

Dealing with exceptions (4)

- ```
  *Main> part2error fn2 (1)
  1
  *Main> part2error fn2 (-1)
  *** Exception: value undefined
  CallStack (from HasCallStack):
    error, called at func1.hs:137:24 in main:Main
  *Main> part2error fn2 (101)
  *** Exception: value undefined
  CallStack (from HasCallStack):
    error, called at func1.hs:137:24 in main:Main
  ```
  ```

functions as partitions

- the gender of x function
  - partitions in males and females
- the ages of x functions
  - some hundred equivalence classes
- surjective $f : A \rightarrow I$
  the relattion R on A
  $aRb \equiv (f(a) = f(b))$
- $R = \{(a, b) \in A^2 | f(a) = f(b)\}$

mapping a function to a equivalence relation

- ```
  fct2equiv :: Eq a => (b -> a) -> b -> b -> Bool
  fct2equiv f x y = (f x) == (f y)

  f :: (b -> a)
  x :: b
  y :: b
  f x :: a
  f y :: a
  (f x) == (f y) :: Bool
  ```
- example

```
*Main> rem 3 2
1
*Main> rem 3 14
3
*Main> fct2equiv (rem 3) 2 14
False
*Main> fct2equiv (rem 3) 2 5
False
```

```
*Main> 2 `rem` 3
2
*Main> 14 `rem` 3
2
*Main> fct2equiv (`rem` 3) 2 14
True
*Main> fct2equiv (`rem` 3) 2 5
True
```
```

backtick

- the backtick (‘ ‘) turns a name to an infix operator
- `a ‘elem‘ b = elem a b`

```
(‘elem‘ b) a = (\x -> x ‘elem‘ b) a
             = a ‘elem‘ b
             = elem a b
```

- `(elem b) a = elem b a`

https://stackoverflow.com/questions/20680779/sections-why-do-i-need-backticks-here

finding equivalence classes

- ```
  block :: Eq b => (a -> b) -> a -> [a] -> [a]
  block f x list = [ y | y <- list, f x == f y ]

  f :: (a -> b)
  x :: a
  list :: [a]
  ```
- examples
  ```
  *Main> block (`rem` 3) 2 [1..30]
  [2,5,8,11,14,17,20,23,26,29]
  *Main> block (`rem` 3) 2 [1 .. 30]
  [2,5,8,11,14,17,20,23,26,29]
  *Main> block (`rem` 3) 1 [1 .. 30]
  [1,4,7,10,13,16,19,22,25,28]
  *Main> block (`rem` 3) 0 [1 .. 30]
  [3,6,9,12,15,18,21,24,27,30]
  *Main> block (`rem` 3) 3 [1 .. 30]
  [3,6,9,12,15,18,21,24,27,30]
  ```
  ```