

1

COMP3006/COMP3906

***Declarative Programming
Languages***

FUNCTIONAL PROGRAMMING

Function Composition

Lecture 23
Tatjana Zrimec

2

Function Composition

- 1 **we can think of a function as a black box that accepts an input and returns an output (or result)**
- 1 **application of a function $f :: a \rightarrow b$**
 - Ø to an input $x :: a$
 - Ø returning output $f\ x :: b$
- 1 **can be represented as follows**

```
graph LR; x --> f[f]; f --> fx[f x]
```

3

Function Composition

1 what happens when we join two such functions?

1 $(f . g)$ is the composite function formed by successively applying g then f to an input

$$(f . g) x = f (g x)$$

∅ also forwards (or left to right) composition with the operator $>.>$

$$(g >.> f) x = (f.g) x = f(g x)$$

4

Type for Function Composition

1 filling in most general types on function boxes

1 shows that the type of function composition is:

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$

∅ applying composition to

$$f :: b \rightarrow c \text{ and } g :: a \rightarrow b$$

∅ yields $f . g :: a \rightarrow c$

Multiple Function Composition

`f . g . h . k`

∅ is single expression composing 4 functions

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

“ no need to write expressions for intermediate values

“ or to introduce local identifiers for them

∅ note: RHS has many parentheses

“ can be avoided in Haskell using the right associative operator `$`

`g $ x = g x`

`f $ g $ x = f (g x)`

`f $ g $ h $ k $ x = f (g (h (k x)))`

Example 1

`f :: Double -> Double`

`f x = log (sin x)`

`= (log . sin) x`

we can drop the x:

`f = log . sin`

∅ `f` is the function that first applies `sin` and then takes the `log` of the (intermediate) result

Example 2

```
g :: Double -> Double
g x  = sin (x ^ 2)
      = (sin . (^2)) x
```

we can drop the x:

```
g      = sin . (^2)
```

$\circ g$ is the function that first squares its input and then takes the `sin` of the (intermediate) result

$\circ (^2)$ is the function that squares its input
here, function application syntax is clearer!

map and filter

- 1 **The basic higher-order functions that allow manipulation of lists.**
- 1 **The map function applies a function to every element of a list, in order to produce a new list.**

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

- 1 **The filter function is used to produce a list of elements that all satisfy a predicate (a Boolean function).**

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs      = [ x | x <- xs, p x ]
```

Example 3

```

type Speller = [String]
toList :: Speller -> [String]
1 sorts the input list of words, then removes
  duplicates by grouping into sublists of identical
  elements and taking the head of each
toList words
  = map head (group (sort words))
1 or, more simply, using function composition
toList = map head . group . sort

```

foldl and scanl

- 1 **The foldl function produces a single value from a list, by folding the elements of that list together, using a function and its null element.**
- 1 **The function is first applied to the null element and the head of the list, and then to that result and the second element of the list, and so on.**

```

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

```

The scanl function produces a list containing all of the intermediate values generated when calculating foldl.

```

scanl          :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs   = q : (case xs of
                        [] -> []
                        y:ys -> scanl f (f q y) ys)

```

Function Composition and Pipes

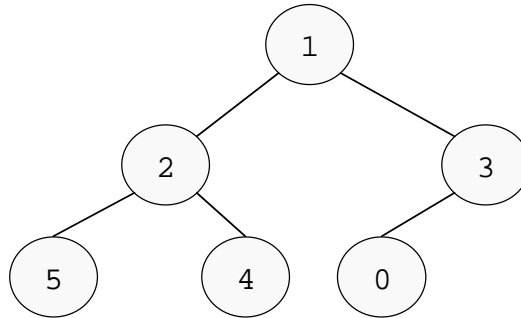
- 1 function composition provides a powerful method for structuring designs: programs are written as a pipeline of operations, passing the appropriate data structures between them.
- 1 encourages composition of behaviours at the function level
 - Ø instead of explicit data manipulation
- 1 operating system shells allow "pipes" between processes
 - Ø standard output of one process can be piped into the standard input of another
- 1 a pipe is a special form of function composition

Binary Trees

- Type
Constructor
- 1 A recursive data type for representing binary trees:


```
data BinTree a = Leaf |
                Node a (BinTree a) (BinTree a)
```
 - 1 A piece of data of type BinTree a is of the form:
 - Ø Leaf alone, representing the terminal tree, or
 - Ø (Node data left right) where
 - data :: a is a data value and
 - left, right :: Tree a each are the tree's left hand and right hand children
 - 1 BinTree is called a *type constructor*, it is used to build new data types from old.
 - 1 Leaf and Node are *data constructors*, they build new data items from old.
- Data
Constructors

A Sample Tree



```

(Node 1( Node 2 ( Node 5 Leaf Leaf )
                ( Node 4 Leaf Leaf ) )
  ( Node 3 ( Node 0 Leaf Leaf ) Leaf ) )
  
```

Simple Functions on *BinTrees*

1 Counting Leaves:

```

countLeaves :: BinTree a -> Integer
countLeaves Leaf = 1
countLeaves (Node _ left right) =
    (countLeaves left) +
    (countLeaves right)
  
```

1 Tree Pruning:

```

pruneTreeAt :: (a -> Bool) -> BinTree a -> BinTree a
pruneTreeAt _ Leaf = Leaf
pruneTreeAt p (Node a left right) =
    if (p a) then Leaf
    else (Node a (pruneTreeAt p left)
           (pruneTreeAt p right))
  
```

the Shape Data Type

- 1 **The Shape data type from Chapter 2 of SOE by Hudak is another example of a variant data type:**

```
data Shape = Rectangle Float Float
           | Ellipse Float Float
           | RtTriangle Float Float
           | Polygon [(Float,Float)]
  deriving Show
```

- 1 **The last line —`deriving Show`— tells the system to build a `show` function for the type `Shape`**

- 1 **We can also define functions yielding refined shapes:**

```
circle, square :: Float -> Shape
circle radius = Ellipse radius radius
square side   = Rectangle side side
```

Functions over Shape

- 1 **Functions on shapes can be defined using pattern matching.**

```
area :: Shape -> Float
area (Rectangle s1 s2) = s1*s2
area (Ellipse r1 r2)   = pi*r1*r2
area (RtTriangle s1 s2) = (s1*s2)/2
area (Polygon (v1:pts)) = polyArea pts
  where polyArea :: [(Float,Float)] -> Float
        polyArea (v2:v3:vs) = triArea v1 v2 v3 +
                                polyArea (v3:vs)
        polyArea _          = 0
```

Note use of auxiliary function.

Note use of nested patterns.

Note use of wild card pattern (which matches anything).

Review of List Comprehension

The zip and unzip

1 **Function zip:**

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _           = []
zip _ []          = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

1 **It has a partner unzip:**

```
unzip :: [(a,b)] -> ([a], [b])
unzip []           = ([],[])
unzip (x,y):zs =
    let (xs,ys) = unzip zs in ( x:xs, y:ys )
```

Generate and Test

- 1 a popular problem solving strategy is called ***generate and test***
- 1 list comprehensions directly implement the generate and test style of computation
- 1 we produce a list of solutions for a problem by
 - Ø ***generating*** a list of candidate solutions
 - Ø ***testing*** the candidates for correctness

Examples of List Comprehension

Ex 1

```
[ n^2 | n <- [0 .. 10], odd n ]
= [1,9,25,49,81]
```

Øequivalent to

```
map (^2) (filter odd [0 .. 10])
```

Ex 2

```
[(x, cos x) | x <- [ 0 , pi/2 .. 4 * pi ], abs (sin x) < 1.0e-6 ]
= [(0.0,1.0), (3.14159,-1.0), (6.28319,1.0), (9.42478,-1.0),
  (12.5664,1.0)]
```

Øequivalent to

```
map f (filter test [ 0 , pi/2 .. 4 * pi ])
  where f x = (x, cos x)
        test x = abs (sin x) < 1.0e-6
```

Examples of List Comprehension

```
[ map toUpper w | w <- words phrase,
                  isUpper (head w) ]

where
  phrase = "G'day mate! Mine are Size 17.
           How big are yours?"
  = ["G'DAY", "MINE", "SIZE", "HOW"]
  Ø equivalent to
  map (map toUpper) (filter test phrase)
  where
    test w = isUpper (head w)
    phrase = "G'day mate! Mine are Size 17.
             How big are yours?"
```

Generate and Test Example

```
-- factors of a
-- positive integer

factors :: Int -> [Int]
factors n = [ f | f <- [1..n],
                n `mod` f == 0
            ]
```

Generate and Test Example

```
-- primes
-- is the list of all prime numbers

primes :: [Int]
primes = [ p | p <- [1 .. ],
            factors p == [1,p]
          ]
```

this generates an infinite list

Haskell computes lazily (on demand) so it's OK

The Set ADT

- 1 **Abstract data types**
 - ∅ name a type
 - ∅ name functions
 - ” constructors, updates, queries
 - ∅ give their type declarations
- 1 **Hide the implementation**
 - ∅ use data or newtype declaration
 - ∅ do NOT export its data constructors

The Set ADT

1 **Some primitive functions**

```
empty    :: Set a
add      :: a -> Set a -> Set a
elemSet  :: a -> Set a -> Bool
```

Specification

1 **sets of items are characterised by the following properties**

- ∅ order in which items are combined is irrelevant
- ∅ duplication of entries is ignored

```
add x (add y set) = add y (add x set)
add x (add x set) = add x set
```

1 **set membership is determined by the following properties**

```
elemSet x empty = False
elemSet x (add y set)
  = (x == y) || elemSet x set
```

More functions

```
singleton :: a -> Set a
```

1 set update operations

```
union      :: Set a -> Set a -> Set a
intersection :: Set a -> Set a -> Set a
difference  :: Set a -> Set a -> Set a
```

1 query operations on sets

```
count    :: Set a -> Int
nullSet  :: Set a -> Bool
subSet   :: Set a -> Set a -> Bool
eqSet    :: Set a -> Set a -> Bool
```

1 conversion between lists and sets

```
fromList :: Eq a => [a] -> Set a
toList   :: Ord a => Set a -> [a]
```

Four implementations

1 **Implementation 1**

∅ as arbitrary lists

∅ toList may expose different representation lists

∅ this is **WRONG!!!**

1 **Implementation 2**

∅ as arbitrary lists

∅ toList is a function on sets

“ it should always returns same list for same set

∅ this implementation is **RIGHT!!**

Four implementations

1 **Implementation 3**

∅ as a sorted list with no duplicates

1 **Implementation 4**

∅ as an ordered binary tree with no duplicates

1 **implementations 2 and 3 all require the list of items to be sorted**

∅ need to introduce class constraint into all the type declarations

“ Ord a =>

∅ allow items to be ordered