LZSCC.212 – Advanced Programming*
Local Bindings, List Comprehension, Zip

Fabio Papacchini
f.papacchini@lancaster.ac.uk

Lancaster University in Leipzig

Academic Year 2024/25

*Content based on Abdessalam Elhabbash and Paul Dempster's slides

Last Week's Content

- Higher-order functions boost flexibility, enabling reusable and expressive code
- play around with foldr, it is a very powerful tool!
- learn to put things together, you know more than you think!
- learn to split your problem into many subproblems
- try things out, the interpreter will stop you as soon as mistake appear

Today's Content

- the higher-order function `foldl`
- local bindings
- list comprehension
- the `zip` function

Today's Content

- the higher-order function `foldl`
- local bindings
- list comprehension
- the `zip` function

Recall foldr

- `foldr` recursively applies a given function to each element of a list
- starts from the rightmost element and accumulate the result

```
sum = foldr (+) 0

-- not actual code, but an idea of how it is executed
  sum [1,2,3]
= foldr (+) 0 [1,2,3]
= foldr (+) 0 (1:(2:(3:[])))
= (1 : (2 : (3 + 0)))
= (1 : (2 + 3))
= (1 + 5)
= 6
```

The Higher-Order Function `foldl`

- `foldl` (fold left) recursively applies a given function to each element of a list
- starts from the leftmost element and accumulate the result

```haskell
-- sum example with a similar behaviour
sum :: Num a => [a] -> a
sum xs = sum' 0 xs
  where
    sum' v [] = v
    sum' v (x:xs) = sum' (v+x) xs
```

The Higher-Order Function `foldl` – Example Behaviour

```haskell
-- function definition
sum :: Num a => [a] -> a
sum xs = sum' 0 xs
  where
    sum' v [] = v
    sum' v (x:xs) = sum' (v+x) xs
```

```haskell
-- function execution
  sum [1,2,3]
= sum' 0 [1,2,3]
= sum' (0+1) [2,3]
= sum' ((0+1)+2) [3]
= sum' (((0+1)+2)+3) []
= (((0+1)+2)+3) = 6
```

The Higher-Order Function `foldl` – Example Behaviour

```haskell
-- function definition
sum :: Num a => [a] -> a
sum xs = sum' 0 xs
  where
    sum' v [] = v
    sum' v (x:xs) = sum' (v+x) xs
```

```haskell
-- function execution
  sum [1,2,3]
= sum' 0 [1,2,3]
= sum' (0+1) [2,3]
= sum' ((0+1)+2) [3]
= sum' (((0+1)+2)+3) []
= (((0+1)+2)+3) = 6
```

This implementation of `sum` it is a simple application of `foldl`. Hence,

```haskell
-- function definition
sum :: Num a => [a] -> a
sum = foldl (+) 0
```

foldl – Definition and Some Derivable Function

foldl can be recursively defined as

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

Example of functions definable via foldl

```
product = foldl (*) 1

or = foldl (||) False

and = foldl (&&) True

reverse = foldl (\acc x -> x:acc ) []
```

Note there are occasions where foldr and foldl produce different results!
Think carefully which one you should use! (e.g., this week workshop)

Today's Content

- the higher-order function `foldl`
- local bindings
- list comprehension
- the `zip` function

Local Bindings

- local bindings are variables or functions defined within a limited scope
- keep code organised, readable, and avoid unnecessary repetition
  - restrict certain definitions to where they are needed
- `let ... in` and `where` allow us to define local bindings in Haskell
  - they might seem similar, but used in different ways

Local Bindings – `let ... in`

- `let` introduces local bindings only available within the `in` expression
- expression-oriented:
  - `let ... in` is an expression it can be used inside other expressions
  - e.g., function bodies, list comprehensions, guards
  - the scope of the bindings is within the `in` expression

```haskell
squareSum :: Int -> Int -> Int
squareSum x y =
  let
    a = x * x
    b = y * y
  in a + b
```

Local Bindings – where

- where introduces local bindings that are available for the entire function body
- definition-oriented:
  - the scope of the bindings is the entire function
  - used to keep the main function clean by placing helper functions or intermediate calculations at the end

```
squareSum' :: Int -> Int -> Int
squareSum' x y = a + b
  where
    a = x * x
    b = y * y
```

Lazy Evaluation

```haskell
-- using let ... in
f x y =
  let
    a = x * y
    b = x - y
  in
    if x > y then b else a

-- using where
f' x y
  | x > y     = b
  | otherwise = a
  where
    a = x * y
    b = x - y

-- examples
> f 5 6      > f' 6 5
30           1
```

Haskell only evaluates expression if needed

- if $x > y$ then a is not computed at all!
- otherwise, b is not computed at all!

E.g., you are allowed to have infinite lists `[0..]`

The next element is only computed if needed

Today's Content

- the higher-order function `foldl`
- local bindings
- list comprehension
- the `zip` function

List Comprehension

In maths, we can define sets via comprehension

$$\{x^2 \mid x \in \{1, 2, 3, 4, 5\}\} = \{1, 4, 9, 16, 25\}$$

In Haskell, a similar notation is used to construct new lists

```
> [ x^2 | x <- [1..5]]
[1,4,9,16,25]
```

In Java, you would do something like. . .

```
List<Integer> squares = new ArrayList<>();
for (int x = 1; x <= 5; x++) {
  squares.add(x * x);
}
```

List Comprehension – Generators

The expression `x <- [1..5]` is called a generator

- it states how to generate values for `x`

Comprehensions can have multiple generators separated by commas

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

The order of generators matters!

```
> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Think of multiple generators as nested loops x's generator first

```
[(x,y) | x <- [1,2,3], y <- [4,5]]
```

In java, x loop first

```
List<int[]> pairs = new ArrayList<>();
for (int x : new int[]{1, 2, 3}) {
  for (int y : new int[]{4, 5}) {
    pairs.add(new int[]{x, y});
  }
}
```

y's generator first

```
[(x,y) | y <- [4,5], x <- [1,2,3]]
```

In java, y loop first

```
List<int[]> pairs = new ArrayList<>();
for (int y : new int[]{4, 5}) {
  for (int x : new int[]{1, 2, 3}) {
    pairs.add(new int[]{x, y});
  }
}
```

Dependent Generators

Later generators can depend on earlier variables introduced by other generators

```
> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

> [(x,y) | x <- [1..3], y <- [(x+1)..3]]
[(1,2),(1,3),(2,3)]
```

This allows for rather powerful lists definitions – concatenation example

```
-- inductive definition
concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ concat xs

-- foldl definition
concat :: [[a]] -> [a]
concat = foldl (++) []
```

```
--list comprehension
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]

-- GHCi
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

List Comprehension – Guards

Guards (Boolean expressions) restrict values produced by earlier generators

```
> [ x | x <- [1..20], even x]
[2,4,6,8,10,12,14,16,18,20]
```

Guards are more powerful than you think!

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

prime :: Int -> Bool
prime n = factors n == [1,n]

primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

```
> factors 15
[1,3,5,15]
> factors 7
[1,7]
> prime 15
False
> prime 7
True
> primes 25
[2,3,5,7,11,13,17,19,23]
```

The order of of generators is of paramount importance!

```
pairsEvenOdd :: [Int] -> [(Int,Int)]
pairsEvenOdd ns = [(x,y) | x <- ns, even x, y <- ns, odd y]

> pairEvenOdd [1..5]
[(2,1),(2,3),(2,5),(4,1),(4,3),(4,5)]

-- wrong definition
pairsEvenOdd ns = [(x,y) | x <- ns, even x, odd y, y <- ns]

error: Variable not in scope: y

-- error as y is not defined for odd y
```

String Comprehension

- a string is a sequence of characters enclosed in double quotes
- internally, however, strings are lists of characters

```
"abc" :: String
-- Same as ['a', 'b', 'c'] :: [Char]
```

Hence, polymorphic functions on lists work on strings!

```
> length "abc"
3
> take 3 "abcde"
"abc"
```

This also means that list comprehension works on lists!

```
count :: Char -> String -> Int
count x xs = length [x' | x' <- xs, x == x']

> count 's' "Mississippi"
4
```

Today's Content

- the higher-order function `foldl`
- local bindings
- list comprehension
- the `zip` function

The zip Function

zip maps two lists to a list of pairs of their corresponding elements

```
> :type zip
zip :: [a] -> [b] -> [(a,b)]

> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

It might not seem much but, again, it is more useful than you think!

Note: the number of generated pairs depends on the length of the shortest list

Checking if an array is sorted

```haskell
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)

sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs xs]

-- GCHi
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]

> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

zip – Example 2

Finding the indices of all occurrences of a given value

```haskell
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [0..], x == x']

-- GCHi
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]

> positions 's' "Mississippi"
[2,3,5,6]
```

Summary

- `foldl` is a potential alternative to `foldr`
- think carefully which one you need, results can be different!
- list comprehension offers a concise way to generate lists
- they are extremely powerful once
  - you get a deep understanding of them
  - you learn to think at a “higher” level
- similar argument holds for zip (but less powerful)

Suggested Readings

Chapter 5 and 7 of "Programming in Haskell" by Graham Hutton