

CS331

Haskell Tutorial 03

A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- List Comprehensions : Zip
- Fold : family of higher order functions that process a data structure in some order and build a return value
- Lazy Evaluation
 - Lazyness example
 - Benefits

The Zip Function

A useful library function is `zip`, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] → [b] → [(a,b)]
```

For example:

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

The Zip Function

Using zip we can define a function returns the list of all pairs of adjacent elements from a list:

```
pairs    :: [a] → [(a,a)]  
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]  
[(1,2), (2,3), (3,4)]
```

The Zip Function

Using pairs we can define a function that decides if the elements in a list are sorted:

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

For example:

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

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

The Zip Function

Using zip we can define a function that returns the list of all positions of a value in a list:

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

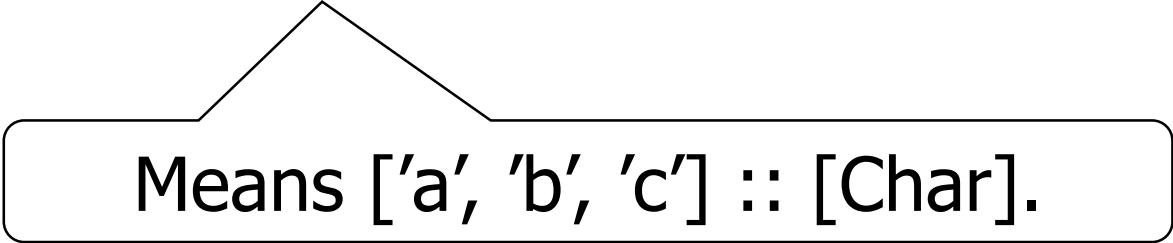
For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

```
"abc" :: String
```



Means ['a', 'b', 'c'] :: [Char].

String Comprehensions

Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```
> length "abcde"
```

```
5
```

```
> take 3 "abcde"
```

```
"abc"
```

```
> zip "abc" [1,2,3,4]
```

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


String Comprehensions

Similarly, list comprehensions can also be used to define functions on strings, such counting how many times a character occurs in a string:

```
count      :: Char → String → Int
count x xs =
    length [x' | x' ← xs, x == x']
```

For example:

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

Is the list sorted?

- Using pairs we can define a function that decides if the elements in a list are sorted:

```
sorted      :: Ord a => [a] -> Bool
```

```
sorted xs  =
```

```
    and [x<= y | (x,y) <- pairs xs]
```

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

➡ True

Positions

- Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions      :: Eq a => a -> [a] -> [Int]
positions x xs =
    [ i | (x',i) <- zip xs [0..n], x == x' ]
    where n = length xs - 1
```

```
positions 0 [1,0,0,1,0,1,1,0]
```

→ [1,2,4,7]

The Foldr function

- A number of functions on lists can be defined using the following simple pattern of recursion:

$$f \ [] = v$$

$$f \ (x:xs) = x \ op \ f \ xs$$

- **f** maps the empty list to some value **v** and any non-empty list to some function **op** applied to its head and **f** of its tail

Right Folding

```
foldr f u [x1,x2,..,xn]
```

```
⇒ f x1 (foldr f u [x2 ..xn])
```

```
⇒ f x1 (f x2 (fold f u [x3..xn]))
```

```
⇒ f x1 (f x2 (... (fold f u [xn]) ...))
```

```
⇒ f x1 (f x2 (... (f xn u) ...)))
```



associate to
right

The Foldr function

- For example

`sum [] = 0`

`sum (x:xs) = x + sum xs`

`product [] = 1`

`product (x:xs) = x * product xs`

`and [] = True`

`and (x:xs) = x && and xs`

The Foldr function

- The higher order function **foldr** (fold right) encapsulates this simple pattern of recursion, with the function **op** and the value **v** as arguments

```
sum          = foldr (+) 0
```

```
product      = foldr (*) 1
```

```
or           = foldr (||) False
```

```
and          = foldr (&&) True
```

The Foldr function

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldr f v [] = v`

`foldr f v (x:xs) = f x (foldr f v xs)`

- However it is best to think of **foldr** non-recursively as simultaneously replacing each `(:)` in a list by a given function and `[]` by a give value

The Foldr function

`sum [1,2,3]` ← **Replace each (:) by (+) and [] by 0**

`= foldr (+) 0 [1,2,3]`

`= foldr (+) 0 (1: (2: (3: [])))`

`= 1 + (2 + (3 + 0)) = 6`

`product [1,2,3]`

`= foldr (*) 1 [1,2,3]` ← **Replace each (:) by (*) and [] by 1**

`= foldr (*) 1 (1: (2: (3: [])))`

`= 1 * (2 * (3 * 1)) = 6`

Left Folding – Tail Recursion

- Accumulate result in a parameter:

```
foldl f u ls =  
  case ls of  
    []      -> u  
    x:xs    -> foldl f (f u x) xs
```

based on accumulation

- What is the type of `foldl`?
- Can we compute `factorial` using it?

Left Folding

```
foldl f u [x1,x2,..,xn]
```

```
⇒ foldl f (f u x1) [x2 ..xn]
```

```
⇒ foldl f (f (f u x1) x2) [x3..xn]))
```

```
⇒ foldl f (f ... (f (f u x1) x2)... xn) []
```

```
⇒ f (... (f (f u x1) x2) ...) xn
```



left is here!

Instance of Left Folding

- Summing a list by accumulation.

```
sumT acc ls =  
  case ls of  
    []      -> 0  
    x:xs    -> sumT (x+acc) xs
```



```
sumList ls = sumT 0 ls
```

```
sumT acc ls = foldl (+) acc ls
```

Iterative List Reversal

- Concatenate first element to last position.

```
revT w [] = w
revT w (x:xs) = revT (x:w) xs
```



Same as:

```
revT w xs = foldl (\ w x -> x:w) w xs
```

What is the time complexity?

Fusion Transformation

- Consider:

```
...sum (double xs) ...  
sum []          = 0  
sum x:xs        = x + (sum xs)  
double []       = []  
double x:xs     = 2*x : (double xs)
```

- Computation reuses smaller functions to build larger ones but may result in unnecessary intermediate structures. They can cause space overheads.
- Solution : Fuse the code together!

Fusion Transformation

- Define:

```
sumdb xs = sum (double xs)
```

- Instantiate: `xs=[]`

```
sumdb []      = sum (double [])  
              = sum []  
              = 0
```

- Instantiate: `xs=x:xs`

```
sumdb x:xs    = sum (double x:xs)  
              = sum (2*x : double xs)  
              = 2*x + sum(double xs)  
              = 2*x + (sumdb xs)
```

Iteration Transformation

- Define:

`sumdbT a xs = a+(sumdb xs)`

- Instantiate: `xs=[]`

`sumdbT a [] = a+(sumdb [])`
`= a`

- Instantiate: `xs=x:xs`

`sumdbT a (x:xs) = a+(sumdb x:xs)`
`= a+(2*x + sumdb xs)`
`= (a+2*x) + (sumdb xs)`
`= sumdbT (a+2*x) xs`

The Composition function

- The library function `(.)` returns the composition of two functions as a single function

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$
$$f \cdot g = \lambda x \rightarrow f (g x)$$

`odd :: Int -> Bool`

`odd = not . even`

Laziness in Haskell

- When the first print statement (“Hello”) happens,
 - The expression myTuple is actually **completely unevaluated**,
 - even though it is defined before the print statement.
- It is represented in memory by what is called a “**thunk**”.
 - Of course, our program knows how to evaluate this thunk when the time comes. But for now, no value is there.
- When it prints the first element of the tuple, it still doesn’t completely evaluate myTuple.

- ```
main = do
 let myTuple = (“first”, map (*2) [1,2,3,4])
 print “Hello”
 print (fst myTuple)
 print (head (snd myTuple))
 print (length (snd myTuple))
 print (snd myTuple)
```

# Laziness in Haskell

- When the compiler sees us call the `fst` function on `myTuple`, it knows `myTuple` must be a tuple.
- Instead of seeing a single thunk at this point, the compiler sees `myTuple` as a **tuple containing two unevaluated thunks**.
- Next, we print the first element of `myTuple`. Printing an expression forces it to be **completely evaluated**. So after this, the compiler sees `myTuple` as a tuple containing a string in its first position and an unevaluated thunk in its second position.

```
main = do
 let myTuple = ("first", map (*2) [1,2,3,4])
 print "Hello"
 print (fst myTuple)
 print (head (snd myTuple))
 print (length (snd myTuple))
 print (snd myTuple)
```

# Laziness in Haskell

- At the next step, we print (head snd myTuple).
- This tells the compiler this second element must be a non-empty list. If it were empty, our program would actually **crash** here.
- This forces the evaluation of this first element (2). However, the remainder of the list **remains an unevaluated thunk**.

```
main = do
 let myTuple = ("first", map (*2) [1,2,3,4])
 print "Hello"
 print (fst myTuple)
 print (head (snd myTuple))
 print (length (snd myTuple))
 print (snd myTuple)
```

# Laziness in Haskell

- Next, we print the length of the list.
- Haskell will do enough work here to determine how many elements are in the list, but it will not actually evaluate any more items.
- The list is now an **evaluated first element**, and **3 unevaluated thunks**.

```
main = do
 let myTuple = ("first", map (*2) [1,2,3,4])
 print "Hello"
 print (fst myTuple)
 print (head (snd myTuple))
 print (length (snd myTuple))
 print (snd myTuple)
```

# Laziness in Haskell

- Finally, we print the full list.
- This evaluates the list in its **entirety**.
  - If we did not do this last step, the final 3 elements would **never be evaluated**.

```
main = do
 let myTuple = ("first", map (*2) [1,2,3,4])
 print "Hello"
 print (fst myTuple)
 print (head (snd myTuple))
 print (length (snd myTuple))
 print (snd myTuple)
```

# Laziness make it faster

- Comparable code in C++ or Java would need to make all three expensive calls to `reallyLongFunction` before calling `F2` with the results.
- But in Haskell, the program **will not call** `reallyLongFunction` until it absolutely needs to.

```
F1 = F2 exp1 exp2 exp3 where
 exp1 = reallyLongFunction 1234
 exp2 = reallyLongFunction 3151
 exp3 = reallyLongFunction 8571
```

```
F2 exp1 exp2 exp3 = if exp1 < 1000
 then exp2
 else if exp1 < 2000
 then exp3
 else exp1
```

# Laziness make it faster

- So in this example, we will always evaluate exp1 in order to determine result of if statement in F2.
- if we happen to have  $\text{exp1} \geq 2000$ , then we'll **never evaluate** exp2 or exp3! We don't need them!
- We'll save ourselves from having to make the expensive calls to reallyLongFunction. As a result, our program will run faster.

```
F1 = F2 exp1 exp2 exp3 where
 exp1 = reallyLongFunction 1234
 exp2 = reallyLongFunction 3151
 exp3 = reallyLongFunction 8571
```

```
F2 exp1 exp2 exp3 = if exp1 < 1000
 then exp2
 else if exp1 < 2000
 then exp3
 else exp1
```



# Demerit of laziness

- When using recursion or recursive data structures, **unevaluated thunks can build up** in heap memory.
- If they consume all your memory, your program will crash.
- In particular, the `foldl` function suffers from this problem. The following usage will probably fail due to a memory leak.  
**`foldl (+) 0 [1..107]`**

# Demerit of laziness

- Laziness can also make it harder to reason about our code.
  - Just because a number of expressions are defined in the same where clause does **not mean** that they will be **evaluated at the same time**.
  - This can easily confuse beginners.

**Thanks**