Session 7: Maps, folds, and type classes (again)

COMP2221: Functional programming

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Recap

- Gave an example of “hidden” complexity in list reversal
- …and one approach to addressing it
- Provided advice on how to approach writing recursive functions “step by step”

Maps and folds

Higher order functions

- We've seen many functions that are naturally recursive
- We'll now look at *higher order functions* in the standard library that capture many of these patterns

Definition (Higher order function)

A function that does at least one of

- take one or more functions as arguments
- returns a function as its result

Higher order functions

- We've seen many functions that are naturally recursive
- We'll now look at *higher order functions* in the standard library that capture many of these patterns

Definition (Higher order function)

A function that does at least one of

- take one or more functions as arguments
- returns a function as its result

- Due to currying, every function of more than one argument is higher-order in Haskell

```
add :: Num a => a -> a -> a
add x y = x + y

Prelude> :type add 1
Num a => a -> a -- A function!
```

This is already a higher order function.

Why are they useful?

- *Common programming idioms* can be written as functions in the language
- *Domain specific languages* can be defined with appropriate collections of higher order functions
- We can use the *algebraic properties* of higher order functions to reason about programs $\Rightarrow$ provably correct *program transformations*

$\Rightarrow$ useful for domain specific *compilers* and automated program generation xs

map f [. . . .]

" don't care what order it happens in".

Could split xs into halves
and then run
map f x_1 ++
map f x_2

This idea lives behind
Hadoop-style mapReduce frameworks.

Higher order functions on lists

- Many *linear recursive* functions on lists can be written using higher order library functions
- `map`: apply a function to a list

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

- `filter`: remove entries from a list

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

- `any`, `all`, `concatMap`, `takeWhile`, `dropWhile`, ...
- For more, see `http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:13`

Function composition

- Often tedious to write brackets and explicit variable names
- Can use *function composition* to simplify this

$$(f \circ g)(x) = f(g(x))$$

- Haskell uses the `(.)` operator

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
-- example
odd a = not (even a)
odd   = not . even -- No need for the a variable
```

- Useful for writing composition of functions to be passed to other higher order functions.
- Removes need to write λ-expressions

pointfree.io

- Called "pointfree" style.

Folds

- *folds* process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

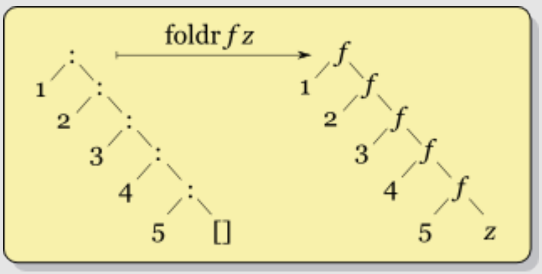
`foldr`: right associative fold

Processes list from the front

linear recursion combining with right-associative f.

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

functools.reduce



Folds

- *folds* process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

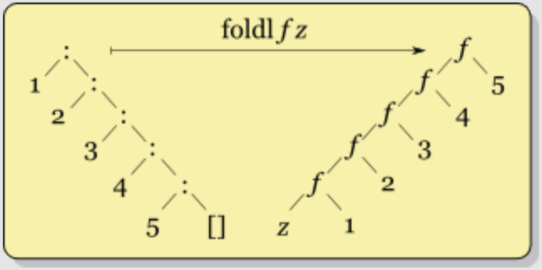
`foldl`: left associative fold

Processes list from the back (implicitly in reverse)

left -associative

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

f.



How to think about this

- `foldr` and `foldl` are recursive
- Often easier to think of them *non-recursively*

foldr

Replace `(:)` by the given function, and `[]` by given value.

```
sum [1, 2, 3]
= foldr (+) 0 [1, 2, 3]
= foldr (+) 0 (1:(2:(3:[])))
= 1 + (2 + (3 + 0))
= 6
```

take 10 (foldr (:) [] [1...])

take 10 (1 : 2 : 3 : ....)

foldl

Same idea, but associating to the left

```
sum [1, 2, 3]
= foldl (+) 0 [1, 2, 3]
= foldl (+) 0 (1:(2:(3:[])))
= (((1 + 2) + 3) + 1)
                  0
= 6
```

(((0 + 1) + 2) + 3)

take 10 (foldl (flip (:)) [] [1...])

Why would I use them?

- Capture many linear recursive patterns in a clean way
- Can have efficient library implementation ⇒ can apply program optimisations
- Actually apply to all **Foldable** types, not just lists
- e.g. `foldr`'s type is actually

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

- So we can write code for lists and (say) trees identically

Folds are general

- Many library functions on lists are written *using folds*

```
product = foldr (*) 1
sum = foldr (+) 0
maximum = foldr1 max
```

max needs at least one entry.

- Practical sheet 4 asks you to define some others

Which to choose?

foldr

requires f is right-associative

- Generally `foldr` is the right (ha!) choice
- Works even for infinite lists!
- Note `foldr (:) [] == id`
- Can terminate early.

xs ++ ys = foldr (:) ys xs

foldl

- Usually best to use *strict* version:

```
import Data.List
foldl' -- note trailing '
```

- Doesn't work on infinite lists (needs to start at the end)
- Use when you *want* to reverse the list: `foldl (flip (:)) [] == reverse`
- Can't terminate early.

flip f a b

flip f = \x y -> f y x

Building block summary

- Prerequisites: none
- Content
  - Introducted definition of *higher order functions*
  - Saw definition and use of a number of such functions on lists
  - Talked about *folds* and capturing a generic *pattern* of computation
  - Gave examples of why you would prefer them over explicit iteration
- Expected learning outcomes
  - student can *explain* what makes a function higher order
  - student can *write* higher order functions
  - student can *use* folds to realise linear recursive patterns
  - student can *explain* differences between `foldr` and `foldl`
- Self-study
  - None

Higher order functions and type classes again

- Saw example higher-order functions on lists
- Now we'll look at *even* more generic patterns
- ...implement our own datatypes
- ...and implement these generic patterns for our datatypes.

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]
```

Separating code and data

- When designing software, a good aim is to hide the *implementation* of data structures
- In OO based languages we do this with classes and inheritence
- Or with *interfaces*, which define a contract that a class must implement

```
public interface FooInterface {
  public bool isFoo();
}

public class MyClass implements FooInterface {
  public bool isFoo() {
    return False;
  }
}
```

- Idea is that *calling* code doesn't know internals, and only relies on interface.
- As a result, we can change the implementation, and client code still works

Generic higher order functions

- In Haskell we can realise this idea with generic *higher order* functions, and type classes
- Last time, we saw some examples of higher order functions for lists
- For example, imagine we want to add two lists pairwise

```haskell
-- By hand
addLists _ [] = []
addLists [] _ = []
addLists (x:xs) (y:ys) = (x + y) : addLists xs ys
-- Better
addLists xs ys = map (uncurry (+)) $ zip xs ys
-- Best
addLists = zipWith (+)
```

import Data.Zip (zipWith)

- If we write our own data types, are we reduced to doing everything "by hand" again?

No: use type classes

- Recall, Haskell has a concept of *type classes*
- These describe interfaces that can be used to constrain the polymorphism of functions to those types satisfying the interface

Example

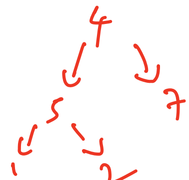
- `(+)` acts on any type, as long as that type implements the **`Num`** interface

  ```
  (+) :: Num a => a -> a -> a
  ```

- `(<)` acts on any type, as long as that type implements the **`Ord`** interface

  ```
  (<) :: Ord a => a -> a -> Bool
  ```

- Haskell comes with *many* such type classes encapsulating common patterns
- When we implement our own data types, we can "just" implement appropriate instances of these classes



```haskell
data [] a = [] | a:[a]
map :: (a -> b) -> [a] -> [b]

data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a)
bmap :: (a -> b) -> BinaryTree a -> BinaryTree b

data RoseTree a = Leaf a | Node a [RoseTree a]
rmap :: (a -> b) -> RoseTree a -> RoseTree b
```

Node a [RoseTree a]

Only difference is the type name of the container. This suggests that we should make a "Container" type class to capture this pattern.

Haskell calls this type class **Functor**

"principled typeclass".

```haskell
class Functor c where
  fmap :: (a -> b) -> c a -> c b
```

[] [a] [b]

If a type implements the **Functor** interface, it is defines structure that we can transform the elements of in a systematic way.

instance Functor [] where
fmap = map.

niftierideology
@niftierideology

Haskell is very simple. Everything is composed of Functads which are themselves a Tormund of Gurmoids, usually defined over the Devons. All you have to do is stick one Devon inside a Tormund and it yields Reverse Functads (Actually Functoids) you use to generate Unbound Gurmoids.

`https://twitter.com/niftierideology/status/1018564372652670976`

Attaching implementations to types

Use an *instance* declaration for the type.

Haskell can derive these instances for you

```
data List a = Nil | Cons a (List a)
  deriving (Eq, Show)
```

Functor

```
instance Functor List where
  fmap _ Nil = Nil
  fmap f (Cons a tail) = Cons (f a) (fmap f tail)

data BinaryTree a = Leaf a | Node a (BinaryTree a) (BinaryTree a)
  deriving (Eq, Show)

instance Functor BinaryTree where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

Generic code

```
list = Cons 1 (Cons 2 (Cons 4 Nil))
btree = Node 1 (Leaf 2) (Leaf 4)
rtree = RNode 1 [RNode 2 [RLeaf 4]]

-- Generic add1
add1 :: (Functor c, Num a) => c a -> c a
add1 = fmap (+1)

Prelude> add1 list
Cons 2 (Cons 3 (Cons 5 Nil))
Prelude> add1 btree
Node 2 (Leaf 3) (Leaf 5)
Prelude> add1 rtree
RNode 2 [RNode 3 [RLeaf 5]]
```

Are all containers Functors?

- It seems like any type that takes a parameter might be a `Functor`
- This is not necessarily the case, we require more than just type-correctness

```
-- A type describing functions from a type to itself
data Fun a = MakeFunction (a -> a)

instance Functor Fun where
  fmap f (MakeFunction g) = MakeFunction id
```

This code type-checks `id :: a -> a` but does not obey the *Functor laws*

1. `fmap id c == c` Mapping the identity function over a structure should return the structure untouched.
2. `fmap f (fmap g c) == fmap (f . g) c` Mapping over a container should distribute over function composition (since the structure is unchanged, it shouldn't matter whether we do this in two passes or one).

How many definitions?

- If I come up with a definition of `fmap` for a type, might there have been another one?
- No! if you can confirm that the functor laws hold

```
fmap id == id
fmap (f . g) == fmap f . fmap g
```

- then you must have written the right thing!

Haskell can't check this for you.

→ Other more sophisticated languages can.

Correctness of `listMap`

```
data List a = Nil | Cons a (List a) deriving (Eq, Show)

instance Functor List where
  fmap _ Nil = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

To show `fmap id == id`, need to show
`fmap id (Cons x xs) == Cons x xs` for any x, xs.

```
-- Induction hypothesis
fmap id xs = xs
-- Base case
 -- apply definition
fmap id Nil = Nil
-- Inductive case
fmap id (Cons x xs) = Cons (id x) (fmap id xs)
== Cons x (fmap id xs)
== Cons x xs -- Done!
```

Exercise: do the same for the second law.

Foldable data structures

- A data type implementing **Functor** allows us to take a container of a's and turn it into a container of b's given a function

  `f :: a -> b`

- **Foldable** provides a further interface: if I can *combine* an `a` and a `b` to produce a new `b`, then, given a start value and a container of `a`s I can turn it into a `b`

deriving Foldable

```
class Foldable f where
  -- minimal definition requires this
  foldr :: (a -> b -> b) -> b -> f a -> b
```

length :: Foldable f => f a -> Int.

Interfaces hide implementation details

- Haskell has *many* type classes in the standard library:
  - `Num`: numeric types
  - `Eq`: equality types
  - `Ord`: orderable types
  - `Functor`: mappable types
  - `Foldable`: foldable types
  - ...
- If you implement a new data type, it is worthwhile thinking if it satisfies any of these interfaces

Rationale

- “abstract” interfaces hide implementation details, and permit *generic* code
- This is generally good practice when writing software
- (I think) the Haskell approach is quite elegant.

Building block summary

- Prerequisites: none
- Content
  - Motivated writing higher order functions for custom data types
  - Recapitulated, and showed more examples, of type classes
  - Saw how implementing type class instances for our data types can make code agnostic to the data structure implementation
  - Saw **`Functor`** and **`Foldable`** type classes, and how they can be used to make new data types behave like builtin ones
- Expected learning outcomes
  - student can *implement* type class instances for new data types
  - student can *describe* some advantages of this approach
- Self-study
  - (Very optional) Chapters 12 & 14 of Hutton's *Programming in Haskell* are an excellent introduction to more of Haskell's "key" type classes