Session 6: Algebraic data types and type classes

COMP2221: Functional programming

Lawrence Mitchell[*]

[*]lawrence.mitchell@durham.ac.uk

Recap

- Discussed and classified types of recursive functions
- Gave an example of “hidden” complexity in list reversal
- Provided advice on how to approach writing recursive functions “step by step”

List comprehensions ✓

Maps and folds

Higher order functions

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

0-order function is just data

1st order is $a \to b$

2nd order is $a \to b \to c$

$(a \to b) \to c$

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

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

Prelude> :type add 1
Num a => a -> a -- A 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 ⇒ provably correct *program transformations*

⇒ useful for domain specific *compilers* and automated program generation

transforms

Map Reduce framework. (Hadoop)

map :: $(a \to b) \to [a] \to [b]$

summaries

reduce :: $(a \to b \to b) \to b \to [a] \to b$

Higher order functions on lists

walk down each element.

- Many *linear recursive* functions on lists can be written using higher order library functions

map even [1,2...]

- map: apply a function to a list

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

=> [False, True, ...]

- filter: remove entries from a list

filter (>10) [1,...]

```
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
- Called "pointfree" style.

map (\x -> not (even x)) [...]

map (not . even) [...]

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

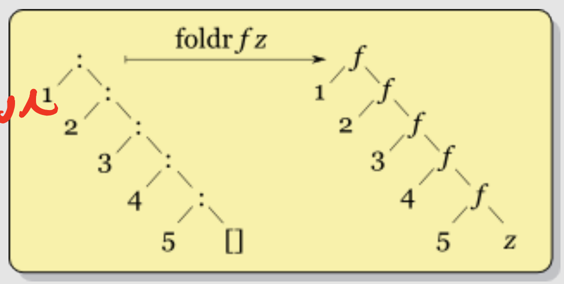
Processes list from the front

:: [a]

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

> initial value

binary right-associative



foldr (:) [] == id

[1, 2, 3] == 1 : 2 : 3 : []

1 `-` 2 `-` 3 `f` z

sum = foldr (+) 0

[1, 2, 3]

1 : 2 : 3 : []

1 + 2 + 3 + 0 = 6

(((( ))

Folds

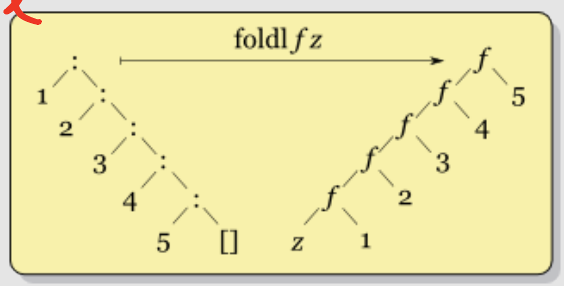
- *folds* process a data structure in some order and build a return value
- Haskell provides a number of these in the standard prelude, with more available in the `Data.List` module

`foldl`: left associative fold

Processes list from the back (implicitly in reverse)

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

f
left-associative
binary operator



$1 : 2 : 3 : []$ $\quad$ foldl f z

$((1 \text{ `}f\text{` } 2) \text{ `}f\text{` } 3) \text{ `}f\text{` } z$

left associative

$(:) :: a \to \overbrace{[a]}^{b} \to \underbrace{[a]}_{b}$

$\text{foldl } (\underline{\text{flip } (:)})\ []$
$== \text{reverse}$

$[a] \to a \to [a]$

$\text{flip} :: (a \to b \to c) \to (b \to a \to c)$

$(\text{flip } (:))\ [2, 3]\ 1$
$= [1, 2, 3]$

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
```

Fine for ∞ lists

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) + 0)
= 6
```

Not fine for ∞ lists

Because need to get to the end to start producing an output.

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
```

- Practical sheet 4 asks you to define some others

Which to choose?

foldr

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

foldl

need left-associativity

- 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.

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

- 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.

```haskell
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]
```

fmap a generic map

"principled typeclass".

```
Prelude> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
Prelude> fmap (*2) [1, 2, 3]
[2, 4, 6]
```

fmap f [...]

(a → b) → [a] → [b]

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

fmap f (Just x)

(a → b) → Maybe a → Maybe b

- Works on any mappable structure
- Should obey *functor laws* (will see example later)

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

Mapping the identity should do nothing

mapping (f ∘ g) should be same as

first mapping g, then mapping f.

fmap changes values inside, not structure.

length (fmap f [1, 2, 3]) = 3

Adding new data types

Defining data types

- It often makes sense to *define* new data types
- Multiple reasons to do this:
  1. Hide complexity
  2. Build new abstractions
  3. Type safety
- Haskell has three ways to do this
  - `type`
  - `data`
  - `newtype` (we won't cover this one)

Type declarations: new names, old types

- A new *name* for an existing type can be defined using a *type declaration*

String as a synonym for the type [Char]

```
type String = [Char]

vowels :: String -> [Char]
vowels str = [s | s <- str, s `elem` ['a', 'e', 'i', 'o', 'u']]

Prelude> vowels "word"
"o"
Prelude> vowels ['w', 'o', 'r', 'd']
"o"
```

- Notice that there is no type distinction: objects of type `String` and `[Char]` are completely interchangeable.

New names, old types II

- We can use these type declarations to make the semantics of our code clearer

An integer position in 2D

```haskell
type Pos = (Int, Int)

origin :: Pos
origin = (0, 0)

left :: Pos -> Pos
left (i, j) = (i - 1, j)
```

- Reader has to expend less brain power to understand the function
- Similar to C's `typedef`

New names, old types III

- Just like function definitions, type declarations can be parameterised over *type variables*

Example

```
type Pair a = (a, a)

mult :: Pair Int -> Int
mult (m, n) = m*n

dup :: a -> Pair a
dup x = (x, x)
```

✗ Can’t use *class constraints* in the definition

✗ Can’t have *recursive* types

Not allowed

```
Prelude> type Tree = (Int, [Tree])
error:
    Cycle in type synonym declarations:
```

Data declarations: new types

- We can introduce a completely *new* type by specifying allowed values using a *data declaration*

A boolean type

```haskell
data Bool = False | True
```

"Bool is a new type, with two new values: False, and True"

- The two values are called *constructors* for the type `Bool`
- Both the type name, and the constructor names, must begin with an upper-case letter.
- This is actually the way `Bool` is implemented in the standard library

Using new types

- Once defined, we can use new types exactly like built in ones

Example

```
data IsTrue = Yes | No | Perhaps

negate :: IsTrue -> IsTrue
-- Pattern matching on constructors
negate Yes     = No
negate No      = Yes
negate Perhaps = Perhaps

Prelude> negate Perhaps
Perhaps
```

Data declarations with fixed type parameters

- The constructors in a data declaration can take arbitrarily many parameters

Example

```haskell
data Shape = Circle Float | Rectangle Float Float
```

“A shape is either a Circle, or a Rectangle. The Circle is defined by one number, the Rectangle by two”

Pattern matching on the constructors:

```haskell
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rectangle x y) = x * y
```

Data declarations with type variables

- We can also make our data declarations *polymorphic* with appropriate type variables

Example

```
data Maybe a = Nothing | Just a
```

"A `Maybe` is either `Nothing` or else a `Just` with a value of arbitrary type"

```
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead (x:_) = Just x
```

Recursive types

- Data declarations can *refer to themselves*

Peano numbers

```haskell
data Nat = Zero | Succ Nat
```

"Nat is a new type with constructors `Zero :: Nat` and `Succ :: Nat -> Nat`"

- This type contains the infinite sequence of values

```haskell
Zero
Succ Zero
Succ (Succ Zero)
...
```

- We could use this to implement a representation of the natural numbers, and arithmetic

```haskell
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
```

Recursive types II

- This kind of recursive type allows very succint definitions of data structures

Linked list

```
data List a = Empty | Cons a (List a)
intList = Cons 1 (Cons 2 (Cons 3 Empty))
== [1, 2, 3]
```

"A List is either Empty, or a Cons of a value and a List"

Linked list in C

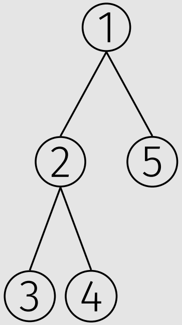
```
typedef struct _Link *Link;
struct _Link {
  void *data;
  Link next;
}
```

A binary tree

A binary tree with values at nodes

```
data BTree a = Empty | Node a (BTree a) (BTree a)
btree = Node 1 (Node 2 (Node 3 Empty Empty)
                       (Node 4 Empty Empty))
               (Node 5 Empty Empty)
```

“A BTree is either Empty, or a Node containing a value and two BTrees”



Pattern matching

- Recall the pattern matching syntax on lists

```haskell
list = [1, 2, 3, 4] == 1:[2, 3, 4]
-- Binds tip to 1, rest to [2, 3, 4]
(tip:rest) = list
```

- The pattern matches the "constructor" of the list, as if the declaration were

```haskell
data [] a = [] | a : [a]
```

- Exactly the same pattern matching applies to data types on their data constructors

```haskell
data List a = Empty | Cons a (List a)
list = Cons 1 (Cons 2 (Cons 3 Empty))
-- Binds tip to 1, rest to (Cons 2 (Cons 3 Empty))
(Cons tip rest) = list
```

Some type theory and contrasts

- Haskell's `data` declarations make *Algebraic data types*
- This is a type where we specify the "shape" of each element
- The two algebraic operations are "sum" and "product"

Definition (Sum type)

An alternation:

```haskell
data Foo = A | B
```

$\{A, B\}$

A value of type `Foo` can either be `A` or `B`

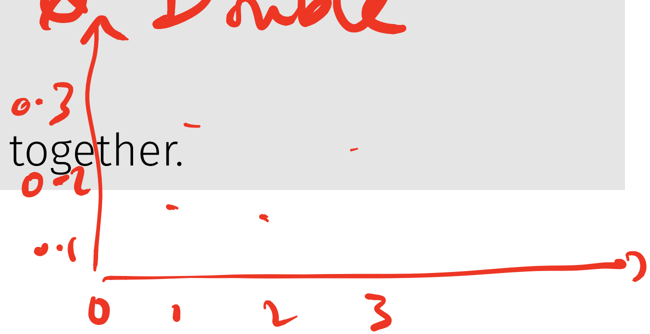
Definition (Product type)

A combination:

```haskell
data Pair = P Int Double
```

Int ⊗ Double

a pair of numbers, an `Int` and `Double` together.



Other languages: product types

- Almost all languages have *product types*. They're just "ordered bags" of things.
- In Python, we can use tuples (or **`namedtuple`**), or classes

Python

```python
pair = (1, 2)
x, y = pair
```

- In C we use structs

C struct

```c
struct Pair {
  int x;
  int y;
}
```

```c
struct Pair p;
p.x = 1;
p.y = 2;
```

- In Java, classes

Other languages: sum types

Haskell: non-exhaustive pattern match

- Useful for type safety/compiler warnings: easy to statically prove that every option is handled
- Less common, although new languages are catching on (e.g. Rust, Swift)
- In C for integers, you can use an `enum`

```
enum Weekdays {
 MON, TUE, WED, THU, FRI, SAT, SUN
};
```

- Not really available properly in Java or Python (you can jump through hoops)
- `https://chadaustin.me/2015/07/sum-types/` is a nice article with more details

Haskell types: pros and cons

inheritance

"O-O"

Classes

- ✓ Easy to add new "kinds of things": just make a subclass
- ✗ Hard to add new "operation on existing things": need to change superclass to add new method and potentially update all subclasses

there is an interface meant to be sealed.

Algebraic data types

- ✗ Hard to add new "kinds of things": need to add new constructor and update all functions that use the data type
- ✓ Easy to add new "operation on existing things": just write a new function

Haskell gets "inheritance" through typeclass interfaces.

Pros and Cons II

interface contract

Adding new things

Just implement a new subclass

```python
class Car(object):
    def seats(self): return 4
class MX5(Car):
    def seats(self): return 2
# Later
class Mini(Car): pass
```

Have to update data constructor

```haskell
data Car = MX5
-- Later
data Car = MX5 | Mini
```

Adding new operations

Must update all classes

abstract method

```python
class Car(object):
    def mpg(self): return 25
    def seats(self): return 4
class MX5(Car):
    def mpg(self): return 30
    def seats(self): return 2
class Mini(Car):
    def mpg(self): return 40
```

Just write new functions

```haskell
seats :: Car -> Int
seats MX5 = 2
seats Mini = 4
mpg :: Car -> Int
mpg MX5 = 30
mpg Mini = 40
```

Building block summary

- Prerequisites: none
- Content
  - Saw how to define new types in Haskell
  - Introduced `type` keyword for synonyms
  - Introduced `data` for completely new types, and the introduction of data constructors
  - Saw pattern matching for data constructors
  - Contrasted sum and product types, and availability in other languages
- Expected learning outcomes
  - student can *define* their own data types
  - student can *explain* difference between `type` and `data`.
- Self-study
  - None

Higher order functions and type classes again

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 (+)
```

- 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

Nomenclature

WARNING!

The *words* class and instance are the same as in object-oriented programming languages, but their *meaning* is very different.

Definition (Class)

A collection of *types* that support certain, specified, overloaded operations called *methods*.

Definition (Instance)

A concrete type that belongs to a *class* and provides implementations of the required methods.

- Compare: type “a collection of related values”
- This is *not* like subclassing and inheritance in Java/C++
- Closest to a combination of Java *interfaces* and *generics*
- C++ “concepts” (in C++20) are also very similar.

Let's look at the types of three "maps"

```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
```

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`

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

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

Attaching implementations to types

Use an *instance* declaration for the type.

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

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

```haskell
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!

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`

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

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