Trees in Haskell

COMP2209 - Programming III

Dr Julian Rathke
Trees of many shapes

- We saw in the previous lecture that we could declare different data types for trees e.g.
  - `data LTree a = Leaf a | Node (LTree a) (LTree a)`
  - `data MTree a = Node a [MTree a]`
- And we see that the trees these produce have different shapes: binary trees etc.
- Algebraic data types are incredibly useful in this respect in that we can specify the shapes of trees very carefully, for example

  ```
  Tree a =
  Leaf a | Node (Tree a)
  | Node (Tree a) (Tree a)
  Tree a =
  Leaf a
  | Node (Node a (Tree a)) (Tree a)
  ```
Red Black Trees

• We also see another benefit from Haskell’s data type declarations that is perhaps less obvious.

• Consider the well-known Red-Black Tree data structure

• These are used as an efficient balanced tree implementation.

• Each node carries an extra bit of information - its colour of either Red or Black.

• There are constraints on how the colours are arranged.
  • The root is Black
  • All leaves are Black
  • If a node is Red its children are Black
  • Every path from any node to a leaf contains the same number of Black nodes
Red Black Trees as a Datatype

• How should we represent this structure:
• How about the following?

```haskell
type Colour = String
data RBTree a = Leaf a Colour
               | Node a Colour (RBTree a) (RBTree a)
```

• But think about how to write the `blackH` function that counts the maximum number of **Black** nodes along any path to a Leaf. i.e. the height of the tree only counting **Black** nodes.

```haskell
blackH (Leaf _ _) = 0
blackH (Node _ c l r) | c == "Red" = maxlr
                       | c == "Black" = 1 + maxlr
where maxlr = max (blackH l) (blackH r)
```
Red Black Trees as a better Datatype

• That was a poor choice - using guards to determine the colour of the nodes could be done much simpler.

• We could use type `Colour = Bool` and codify Black as True and Red as False
  • This would allow us to pattern match when defining functions over RBTrees
  • But it would not be very transparent for readability.

• Better, is to use appropriately named Constructors and pattern match on them.

```haskell
data RBTTree a = Leaf a
  | RedNode a (RBTTree a) (RBTTree a)
  | BlackNode a (RBTTree a) (RBTTree a)
```

```haskell
blackH (Leaf _ _) = 0
blackH (RedNode _ l r) = maxlr
blackH (BlackNode _ l r) = 1 + maxlr
where maxlr = max (blackH l) (blackH r)
```
Constructors as Node Types

• This suggests an interesting use of Tree datatypes whereby different node types form part of the structure of the type.

• This is incredibly useful when it comes to modelling more complex tree types.

• For example, consider Abstract Syntax Trees for a programming languages.

• These are trees that represent the structure of a program expression and remove unnecessary syntactic information (e.g. parentheses).

• Different node types can represent different types of expressions directly in the tree structure.

• This makes them an excellent data structure for then defining functions of programs (such as an interpreter) over.
AST for Arithmetic Expressions

- Let’s do this for a simple language of arithmetic expressions:

```haskell
data Expr = Val Int | Add Expr Expr | Sub Expr Expr
```

- This data type just looks like a grammar for a language doesn’t it?
- That is no accident.
- The values of this data type are ASTs for the language.
- We can write the interpreter function to evaluate expressions very easily.

```haskell
eval :: Expr → Int

eval (Val n) = n

eval (Add e1 e2) = eval e1 + eval e2

eval (Sub e1 e2) = eval e1 - eval e2
```
Evaluating Propositions

• Let’s build a data type to represent ASTs of propositional logic formula.

```haskell
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

• Trees in this type are either leaf nodes containing constant booleans or propositional variables

• Or internal nodes with a single child (Not) or two children (And, Imply)
Evaluating Propositions

- Let’s try write an evaluation function for the Prop type then:

```haskell
eval :: Prop → Bool

eval (Const b) = b

eval (Var c) = ???

eval (Not p) = not $ eval p

eval (And p q) = eval p && eval q

eval (Imply p q) = eval p <= eval q
```

- We need to provide values for each of the propositional variables that exist in the formula.

- This is what is known as a substitution.
Evaluating Propositions

- Define type \( \text{Subst} = \text{Assoc} \ \text{Char} \ \text{Bool} \)
- where type \( \text{Assoc} \ k \ v = [ (k,v) ] \)
- and \( \text{find} \ x \ t = \text{head} \ [ v \mid (k',v) \leftarrow t, k==k' \] \)

Then eval simply carries a substitution around in order to interpret variables, whereby it uses find.

\[
\text{eval} :: \text{Subst} \to \text{Prop} \to \text{Bool} \\
\text{eval} \ s \ (\text{Const} \ b) = b \\
\text{eval} \ s \ (\text{Var} \ c) = \text{find} \ c \ s \\
\text{eval} \ s \ (\text{Not} \ p) = \text{not} \ \$ \ \text{eval} \ s \ p \\
\text{eval} \ s \ (\text{And} \ p \ q) = \text{eval} \ s \ p \ \&\& \ \text{eval} \ s \ q \\
\text{eval} \ s \ (\text{Imply} \ p \ q) = \text{eval} \ s \ p \ \leq\leq \ \text{eval} \ s \ q
\]
Higher-order functions and Trees

- We’ve seen that the combination of pattern matching and recursion makes it easy to define functions over structured data.
- We can also make use of higher-order functions over structured data.
- The examples of higher-order functions we have seen already have mostly been concerned with list processing
  - map, filter, fold
- There is no reason that we can’t extend these concepts to other structures.
- Let’s consider a Tree datatype and map
Tree maps

• Suppose we wish to take a tree of Strings and map the function upper that converts a string to upper case across all nodes in the tree.

• We could write a recursive function using pattern matching to do so but we could also recognise that this function would fit a well-known template of recursion we know loosely as map.

• We can easily define this for the following datatype

```haskell
data LTree a = Leaf a | Node (LTree a) (LTree a)

lTMap :: (a -> b) -> LTree a -> LTree b
lTMap g (Leaf x) = Leaf (g x)
lTMap g (Node l r) = Node (lTMap g l) (lTMap g r)
```
Other Tree maps

But what about the different Tree types? How about

```haskell
data ITree a = Leaf | Node a (ITree a) (ITree a)

iTMap :: (a -> b) -> ITree a -> ITree b
iTMap g (Leaf) = Leaf
iTMap g (Node x l r) = Node (g x) (iTMap g l) (iTMap g r)

Or

data MTree a = Node a [MTree a]

mTMap :: (a -> b) -> MTree a -> MTree b
mTMap g (Node x ts) = Node (g x) (map (mTMap g) ts)
```
Introducing Functors

• The type of the “map” function in each case above is pretty much the same except for the name of the Tree type.

• Obviously the implementation changes each time to reflect the different structure.

\[
\text{map} :: (a \rightarrow b) \rightarrow \text{Tree } a \rightarrow \text{Tree } b
\]

• This suggests we could formulate the map function in a class of mappable types.

• This would give us a consistent name for the map function.

• The concept of transformations of the form

\[(a \rightarrow b) \rightarrow (f \; a \rightarrow f \; b)\]

for some function \(f\) on types is actually well known from Mathematics and is called a \textbf{Functor}.
Class Functor

- It is a simple class with just one function:

```haskell
class Functor f where
    fmap :: (a → b) → f a → f b
```

- Here are some instances:

```haskell
instance Functor [] where
    fmap = map

instance Functor MTree where
    fmap = mTMap

instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap g (Just x) = Just (g x)
```

All called fmap
Expected Properties of fmap

• In order to preserve the structure of data being traversed during a fmap, it is expected that Functors will satisfy certain behaviours.

• In particular the following equations should hold for any implementation of fmap

• This is not checked by the compiler - it is down to you as a good programmer to verify these properties yourself.

• We’ll study how to do this in a later lecture.

Functor Law 1: \( \text{fmap id} = \text{id} \)

Functor Law 2: \( \text{fmap (g.h)} = \text{fmap g . fmap h} \)

This says that mapping a function with no effect should have no effect on the structure.
Moving around a Tree

• In Haskell data structures have no identity - in the sense that we cannot address the memory location in which they are stored
  • i.e. No pointers, no object references
• Suppose we have a Tree structure and want to modify some part of it - how do we address the substructure to change?
• Should we use some sort of path description?
  • cf. a full path name of a file in a file system
• Should we use some sort of relative addressing?
  • cf. move up a directory, enter a subdirectory
• The basic Tree datatype in Haskell shown above doesn’t support either of these. Let’s look at how to improve it so that it does.
Giving Directions

• We’ll just work with the Tree datatype as follows:

```haskell
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

• To specify a path in a tree to any node it is sufficient to use a list of Left/Right directions relative to the root node.

```haskell
data Direction = L | R
type Directions = [Direction]
```

• We can then use these to navigate a Tree

```haskell
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```
Leaving a Trail

• Following a path is all well and good but it doesn’t tell us where we are in the structure.
  • cf. the pwd Unix shell command.

• Let’s call our Directions type a Trail instead and write functions that let us follow directions and record the trail we have traversed.

```haskell
type Trail = [Direction]

goLeft :: (Tree a, Trail) → (Tree a, Trail)
goLeft (Node _ l _ , ts) = (l , L:ts)
goRight :: (Tree a, Trail) → (Tree a, Trail)
goRight (Node _ _ r , ts) = (r , R:ts)
```
Following a Trail

• This mean we can specify a path by using a sequence of goLeft/goRight function calls.

• It is useful to define the backwards application operator
  
  \[ x \rightarrow f \equiv f \ x \]

  to write paths in a neat way.

\[
\text{myTree} = \text{Node 5 ( Node 4 (Node 2 \ldots \ldots etc)
myTree \rightarrow \text{goLeft} \rightarrow \text{goLeft} \rightarrow \text{goRight etc}
\]

So we can use this to nicely navigate downwards through a tree. But what about travelling back up a tree?

As we travel left or right to a subtree we throw away the other subtree that we don’t visit. We need to keep this information in order to reconstruct any parent node.
Reconstructing an upwards Trail

• To travel upwards in a tree then we need to rebuild the parent node with the value stored in it and its two subtrees. Our trails must be redefined to store this information.

data Direction a = L a (Tree a) | R a (Tree a)
type Trail a = [Direction a]

goLeft, goRight, goUp :: (Tree a, Trail a) → (Tree a, Trail a)
goLeft (Node x l r, ts) = (l, L x r:ts)
goRight (Node x l r, ts) = (r, R x l:ts)
goUp (t, L x r : ts) = (Node x t r, ts)
goUp (t, R x l : ts) = (Node x l t, ts)
Zippers

- This technique of pairing a substructure and means of reconstructing the larger structure around it is well-known in functional programming and is called a Zipper type.

```haskell
data Direction a = L a (Tree a) | R a (Tree a)
type Zipper a = (Tree a, Trail a)
```

- You can write some simple Tree functions using them: e.g.

```haskell
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, ts) = (Node (f x) l r, ts)
modify f (Empty, ts) = (Empty, ts)
```

Apply f to the current Node value

```haskell
attach :: Tree a -> Zipper a -> Zipper a
attach t ( _ , ts) = (t, ts)
```

Replace current Node with subtree t

```haskell
goRoot :: Zipper a -> Zipper a
goRoot ( t , [] ) = (t, [])
goRoot z = goRoot ( goUp z )
```

Travel back to root
YOUR QUESTIONS

Next Lecture:
Graphs in Haskell