Data Structures: Binary Trees

Winter 2016-2017

Table of Contents

Binary Trees

Binary Trees

We define simple binary trees as follows:

```
data BinTree a = Nil
               | Node a (BinTree a) (BinTree a)
  deriving (Show)

-- examples

tree1 :: BinTree Char
tree1 = Node 'a' Nil (Node 'b' Nil Nil)

tree2 :: BinTree Int
tree2 = Node 10
          (Node 5 Nil Nil)
            (Node 13 Nil Nil)
```

Adding to a given BinTree

One obvious operation is to insert a new element in the tree. Here we assume that the BinTree is ordered according to the binary search tree property:

- All labels in the left-subtree are smaller than the label at the root.
- All labels in the right-subtree are larger than or equal to the label at the root.

```
btinsert x Nil = Node x Nil Nil
btinsert x (Node y l r)
  | x < y   = Node y (btinsert x l) r
  | x >= y  = Node y l (btinsert x r)
```

This should be familiar. We insert the new element on the left or right, depending on how it compares to the root of the tree.

Converting a list to a tree

Given a list of elements, we can construct a BinTree by repeatedly inserting the elements from the list into the tree. Here's an easy recursive version:

```
insertall0 :: Ord a => BinTree a -> [a] -> BinTree a
insertall0 t [] = t
insertall0 t (e:es) = insertall0 (btinsert e t) es
```

We'd call it this way:

```
Main> insertall0  Nil   [3,2,5,4,7,6]
Node 3 (Node 2 Nil Nil)
       (Node 5 (Node 4 Nil Nil)
               (Node 7 (Node 6 Nil Nil)  Nil))
```

I've indented the output to give a sense of the structure of the tree.

Developing insertall :Step 1

As defined previously, insertall applies btinsert repeatedly to a list of elements.
In other words, it's "folding" the list using btinsert .

```
insertall2 :: Ord a => BinTree a -> [a] -> BinTree a
insertall2 t l = foldr btinsert t l
```

Recall: The function **foldr** takes an operator f , and applies it to a list of elements.

Developing insertall :Step 2

To simplify, we can leave the 2 arguments to insertall2 implicit, and define very simply:

```
insertall :: Ord a => BinTree a -> [a] -> BinTree a
insertall = foldr btinsert
```

Be sure to understand this. The summary that **foldr** is creating happens to be a BinTree, by adding each element to it.

Using Nil for the first argument to insertall

```
Main> insertall Nil [3,2,5,4,7,6]
Node 6 (Node 4 (Node 2 Nil (Node 3 Nil Nil))
                (Node 5 Nil Nil))
       (Node 7 Nil Nil)
```

or a non-empty tree:

```
Main> insertall (Node 5 Nil Nil) [3,2,5,4,7,6]
Node 5 (Node 4 (Node 2 Nil (Node 3 Nil Nil))
               Nil)
       (Node 6 (Node 5 Nil Nil)
               (Node 7 Nil Nil))
```

Defining **map** for BinTrees

We have already seen **map** for lists.

Here is the equivalent version written especially for BinTree.

It applies a function f to every label in the tree:

```
btmap :: (a -> b) -> (BinTree a) -> (BinTree b)
btmap _ Nil = Nil
btmap f (Node x l r) = Node (f x) (btmap f l) (btmap f r)
```

Application of BinTrees

Here we define a simple employee record.

```
data Employee = Record String [Int]
  deriving (Show)
```

It's convenient to provide accessors for this type:

```
getName (Record n p) = n
getList  (Record n p) = p
```

Making an instance of **Eq**

To compare employee records for equality, we have to make the type an instance of **Eq**:

```
instance Eq Employee where
  (Record n l) == (Record m p) = n == m
```

This is necessary before we deal with the **Ord** class.

Making an instance of Ord

If we want to put them into a binary tree, we need to make it an instance of **Ord** as well:

```
instance Ord Employee where
  (Record n l) <= (Record m p) = n <= m
```

A simple example tree

Now we can show a simple example of putting employee records into trees:

```
tree3 = insertall Nil
          [Record "cam" [2009]
          ,Record "bet" [2010]
          ,Record "al"  [2008]
          ]
```

Note the use of the function `insertall`

Promotions for employees

This function takes the year, and an employee, and adds the year into the promotions list:

```
addpromotion :: Int -> Employee -> Employee

addpromotion x (Record name promotions)
  = Record name (x:promotions)
```

(it's just a teaching example; it's not supposed to be realistic)

Applying a function conditionally

We'd like to promote one or more employees based on a certain condition.

The following function tests if a value `e` satisfies a condition `p`, and if so, applies `f` to it.

```
oncondition p f e = if (p e) then (f e) else e
```

What type is `oncondition`?

Promoting AI the hard way

To promote an employee, we want to find that employee in the tree, and put the promotion in the list.

```
findPromote nm yr Nil = Nil
findPromote nm yr (Node n l r)
   | (getName n) == nm = Node (addpromotion yr n) l r
   | (getName n) < nm = Node n (findPromote nm yr l) r
   | (getName n) > nm = Node n l (findPromote nm yr r)
```

And we use it this way:

```
Main> findPromote "al" 2012 tree3
Node (Record "al" [2012,2008]) Nil
          (Node (Record "bet" [2010]) Nil
      (Node (Record "cam" [2009]) Nil Nil))
```

Drawback: for every conceivable update operation, we need a special purpose recursive function.
It's fragile (what if your data type changes?), and tedious.
There is a better way.

Are you Al?

- We need a way to ask if an employee's name is "al"

```
empNamed name (Record n l) = (n == name)
```

- Or we could use the accessor:

```
empNamed name emp = ((getName emp) == name)
```

- This is just function composition, again!

```
empNamed name emp = ((== name) . getName) emp
```

- Simplifying, by leaving emp implicit:

```
empNamed name = (== name) . getName
```

Promoting Al the Haskell way

We manipulate functions the way other languages manipulate other data types.

We already have `btmap` which does something to every element in the tree.

Let's just apply `addpromotion` to every element in the tree, on the condition that the employee's name is Al.

Here it is in Haskell:

```
Main> btmap (oncondition ((== "al").getName)
                                  (addpromotion 2017)) tree3
Node (Record ''al''  [2017,2008])  Nil
    (Node (Record ''bet''  [2010])  Nil
        (Node (Record ''cam''  [2009])  Nil  Nil))
```

(I added a newline again, in the expression, to keep everything on the page)

We constructed the first argument to btmap "on the fly".

We manipulated functions in the same kind of way we manipulate numbers.

Making things slightly easier

Let's define a function to promote employees for any kind of reason.

```
promote cond year = btmap (cond (addpromotion year))
```

Notice the function composition there? Rewrite!

```
promote cond = btmap . cond . addpromotion
```

An example

This year, let's promote everyone who hasn't been promoted since 2010:

```
Main> promote (oncondition ((<2010).head.getList)) 2017 tree3
Node (Record "al" [2017,2008]) Nil
          (Node (Record "bet" [2010]) Nil
              (Node (Record "cam" [2017,2009]) Nil Nil))
```

To describe who is to be promoted, we have ((<2010).**head**.getList), which says:

- Given an employee record
- grab the list of promotions
- look at the first element
- check if it was before 2010

(If this is still unclear to you, use the lambda-notation and the definition of (.) to see what ((<2010).**head**.getList) evaluates to.)

Getting back some perspective

The point of these examples:

- To deepen our understanding of the way functions are built out of other functions
- To see that higher order functions can be put to many uses, by remaining flexible.
- Advanced Haskell programmers derive implementations by manipulating functions the way we manipulate numbers in arithmetic.