User defined data types - part 2

Videos

The following videos are also linked at appropriate points of this handout for your convenience.

- Binary search trees (27min).
- Rose trees (13min).
- Game trees (6min).
- Permutation trees (25min).
- Expression trees (13min).

Total time 1:24.

Contents

Experimenting with the Haskell code included here

You should experiment with the Haskell code in theses notes in order to achieve full understanding. This means running the code and adding things to it, such as solutions of exercises and puzzles, or your own brilliant ideas. The code is available at data2.hs, or can be obtained from this handout with `mdtohs` as in the previous handout. Make your own **copy** of this file to avoid conflicts when we update it.

Binary search trees

A video discussing the next few sections is available on Canvas

Now let's consider binary *search* trees.

Operations on binary search trees

We import the Haskell code from the previous handout:

```haskell
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}

import Data1
import System.Random
```

The following checks whether a binary tree (containing values of some *ordered* type) is a binary search tree:

```
isBST :: Ord a => BT a -> Bool
isBST Empty        = True
isBST (Fork x l r) = allSmaller x l
                  && allBigger  x r
                  && isBST l
                  && isBST r

allSmaller :: Ord a => a -> BT a -> Bool
allSmaller x Empty        = True
allSmaller x (Fork y l r) = y < x
                         && allSmaller x l
                         && allSmaller x r

allBigger :: Ord a => a -> BT a -> Bool
allBigger x Empty = True
allBigger x (Fork y l r) = y > x
            && allBigger x l
            && allBigger x r
```

This is not an efficient way of checking the binary search tree property (it runs in quadratic time), but is very close to the English-prose definition you know.

We get

```
   isBST btexample = True
   isBST btleft = True
   isBST (mirror btexample) = False
```

A better way of checking the BST property (in linear time) is to produce its in-order traversal. It turns out that a tree has the BST property if and only if its in-order traversal is a sorted list (one can prove this by induction on trees).

```
isBST' :: Ord a => BT a -> Bool
isBST' t = isIncreasing(treeInOrder t)

isIncreasing :: Ord a => [a] -> Bool
isIncreasing []       = True
isIncreasing (x:[])   = True
isIncreasing (x:y:zs) = x < y && isIncreasing(y:zs)
```

Puzzle. Can you write another version of isBST that runs in linear time *without* producing the in-order traversal list as an intermediate step?

As you will remember, the point of binary search trees is that they can be searched fast (logarithmic time) if they are sufficiently balanced:

```
occurs :: Ord a => a -> BT a -> Bool
occurs x Empty        = False
occurs x (Fork y l r) = x == y
                     || (x < y && occurs x l)
                     || (x > y && occurs x r)
```

For the next function, we want that if the input is a BST, then so is the output.

```
insert :: Ord a => a -> BT a -> BT a
insert x Empty                    = Fork x Empty Empty
insert x (Fork y l r) | x < y     = Fork y (insert x l) r
                      | x > y     = Fork y l (insert x r)
                      | otherwise = Fork y l r
```

If the element to be inserted is already in the tree, we give the same tree as a result (in the `otherwise` case), as our BST's are not allowed to have repetitions (according to the definition of isBST). Of course, there are alternative ways, such as using a `Maybe` return type to indicate with `Nothing` that the element is already there:

```
insert' :: Ord a => a -> BT a -> Maybe(BT a)
insert' x Empty                    = Just(Fork x Empty Empty)
insert' x (Fork y l r) | x < y      = case insert' x l of
                                        Nothing -> Nothing
                                        Just l' -> Just(Fork y l' r)
                       | x > y      = case insert' x r of
                                        Nothing -> Nothing
                                        Just r' -> Just(Fork y l r')
                       | otherwise = Nothing
```

This code can be slightly simplified using the fact that `Maybe` is a monad, but we'll come to that later.

The most difficult function for BST's is deletion:

```
delete :: Ord a => a -> BT a -> BT a
delete x Empty = Empty -- or you may prefer undefined (and even Nothing)
delete x (Fork y l r) | x < y                  = Fork y (delete x l) r
                      | x > y                  = Fork y l (delete x r)
                      | x == y && l == Empty = r
                      | x == y && r == Empty = l
                      | otherwise              = Fork (largestOf l) (withoutLargest l) r

largestOf :: Ord a => BT a -> a
largestOf Empty              = undefined
largestOf (Fork x l Empty) = x
largestOf (Fork x l r)      = largestOf r

withoutLargest :: Ord a => BT a -> BT a
withoutLargest Empty              = undefined
withoutLargest (Fork x l Empty) = l
withoutLargest (Fork x l r)      = Fork x l (withoutLargest r)
```

- Can you write a function `delete'` with a `Maybe` return type to indicate that there is nothing to delete?
- Can you combine the last two functions `largestOf` and `withoutLargest` into a single one, using a pair type as the result, so as to get a more efficient version of the delete function? And then this together with `Maybe` rather than using `undefined` ?

Testing and experimenting with the BST code

It is very important for you to learn how to test your code (for correctness) and experiment with it (for efficiency), both for this module and for your professional life as a software developer, if this is your chosen career path after you finish your degree. Here are some starting ideas for you to design your own tests for your work for this module. The main one is that you write code to test that what should be true is indeed true, and to test the run times.

Later we will meet the random monad. For the moment we work with an infinite list of pseudo random integers (needs `import System.Random` at the top of a Haskell file):

```
randomInts :: [Int]
randomInts = randomRs (minBound,maxBound) (mkStdGen seed)
             where seed = 42
```

We can insert many elements in a tree as follows:

```
inserts :: Ord a => [a] -> BT a -> BT a
inserts []     t = t
inserts (x:xs) t = inserts xs (insert x t)
```

Then we define data for our testing and experimentation:

```
aBigBST :: BT Int
aBigBST = inserts (take (10^6) randomInts) Empty

itsHeight = height aBigBST
itsSize   = size aBigBST
itsBST    = isBST aBigBST
itsBST'   = isBST' aBigBST
```

Now we test and experiment with this as follows in `ghci` (where in practice we should have created a file with code for this):

```
*Main> :set +s -- ask ghci to print time and space usage
*Main> itsHeight
49
(20.66 secs, 11,618,092,240 bytes)
*Main> itsHeight      -- again
49
(0.01 secs, 0 bytes)  -- fast because it got stored (part of what laziness is)
*Main> itsSize
1000000
(0.50 secs, 248,066,824 bytes) -- fast because the tree is already computed
*Main> itsBST
True
(12.84 secs, 8,691,110,224 bytes) -- slow because of inefficient algorithm
*Main> itsBST'
True
(1.10 secs, 1,198,200,632 bytes) -- the alternative algorithm is much more efficient
*Main>
```

Notice the height is not optimal (that would be `log2(10^6)`, which is approximately 20), but it is higher (namely 49), meaning that searches in that tree will take at most 49 steps. Nevertheless, the search operation will be very fast, even when what we are looking for is not in the tree:

```
*Main> occurs 17 aBigBST
False
(0.01 secs, 0 bytes)
```

Delete many elements from a tree:

```
deletes :: Ord a => [a] -> BT a -> BT a
deletes []     t = t
deletes (x:xs) t = deletes xs (delete x t)
```

Delete half of the elements we inserted:

```
aSmallerTree :: BT Int
aSmallerTree = deletes (take (5 * (10^5)) randomInts) aBigBST
```

This tree is not computed until it is used, because of lazy evaluation. The following forces all deletions to be performed.

```
*Main> height aSmallerTree
45
(8.88 secs, 5,515,246,520 bytes)
```

Then, as a sanity check, we test whether our deletion algorithm didn't destroy the BST property:

```
*Main> isBST' aSmallerTree
True
(0.52 secs, 583,510,072 bytes)
```

Good.

```
evenBigger :: BT Int
evenBigger = inserts (take (10^7) randomInts) Empty
```

Oops:

```
*Main> height evenBigger
*** Exception: stack overflow
```

There is an option to increase the size of the stack when you run ghci:

```
$  ghci data.hs +RTS -K1G -RTS
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( data.hs, interpreted )
Ok, modules loaded: Main.
*Main> :set +s
*Main> height evenBigger
58
(269.24 secs, 134,999,934,888 bytes)
*Main> 269.24 / 60
4.487333333333334
*Main>
```

That's 4.49 minutes on my machine. But, once the tree is computed the first time, it stays computed, and we get e.g.

```
*Main> height evenBigger
58
(4.24 secs, 2,506,734,080 bytes)
*Main> occurs 17 evenBigger
False
(0.01 secs, 92,952 bytes) -- magic? :-)
```

The last expression would take a long time if executed immediately after loading the file, before the tree is computed:

```
*Main> isBST' evenBigger
True
(15.38 secs, 13,271,069,712 bytes)
```

So we have a BST with 10,000,000 elements, of height 58, which can hence be searched fast. We can easily get bigger trees. Now consider

```
fullBST :: Integer -> Integer -> BT Integer
fullBST x y | x == y    = Fork x Empty Empty
            | x+1 == y  = Fork y (Fork x Empty Empty) Empty
            | x+1 <  y  = Fork m (fullBST x (m-1)) (fullBST (m+1) y)
            | otherwise = undefined
  where m = (x + y) `div` 2
```

What does this function do? Convince yourself that if `x<=y` then `treeInOrder(fullBST x y) = [x..y]`. For instance

```
    treeInOrder(fullBST 2 11) = [2,3,4,5,6,7,8,9,10,11]
```

In the following, the same tree gets computed (to the extent that it is needed) every time because it is not assigned to a variable:

```
*Main> occurs 17 (fullBST 1 (10^8))
True
(0.01 secs, 0 bytes)
*Main> occurs 17 (delete 17 (fullBST 1 (10^8)))
False
(0.01 secs, 0 bytes)
*Main> height (fullBST 1 (10^8))
27
(123.90 secs, 63,316,811,920 bytes)
*Main> 123.90 / 60
2.065
```

This amounts to two minutes. To figure out that 17 is (not) in the above trees, it is not necessary to build the whole tree (Haskell is lazy), and this is why this is faster than computing the height.

Finally:

```
*Main> deletes (take (10^5) randomInts) (inserts (take (10^5) randomInts) Empty)
Empty
(2.74 secs, 1,686,246,488 bytes)
*Main> deletes (take (10^6) randomInts) (inserts (take (10^6) randomInts) Empty)
Empty
(42.25 secs, 20,813,980,136 bytes)
```

BST sort, quick sort and merge sort

Of course, as you know, you can sort lists using this, but it will remove repetitions because binary search trees don't allow for repetitions (accounted for by our version of `insert` and `inserts`):

```
bstsort :: Ord a => [a] -> [a]
bstsort xs = treeInOrder(inserts xs Empty)
```

A form of quick sort is easy to write:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [l | l <- xs, l < x]
            ++ [x]
            ++ qsort [r | r <- xs, r >= x]
```

You can easily modify this to remove duplicates in the resulting sorted list (try it). Merge sort can be defined as follows, where we split a list in even and odd positions, rather than first half and second half, for the sake of simplicity and efficiency:

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] [] = []
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys

eosplit :: [a] -> ([a],[a])
eosplit []       = ([],[])
eosplit [x]      = ([x],[])
eosplit (e:o:xs) = case eosplit xs of
                     (es,os) -> (e:es, o:os)

msort :: Ord a => [a] -> [a]
msort xs | length xs <= 1  =  xs
         | otherwise       = merge (msort es) (msort os)
                             where (es, os) = eosplit xs
```

As you know, quick sort for sorted or reverse-sorted lists is slow (quadratic time). So don't bother to try e.g.

```
    sum(qsort [1..(10^5)])
```

as it would take `(10^5)^2` steps, which amounts to `10^10`, or 10 billion, steps. The same will happen with `bstsort`. However, for random lists they work better, with `qsort` faster than `bstsort`. Recall that `msort` is always `n * log n`. We haven't tested `msort` for the sake of brevity (these notes are already long as they are), but you may wish to do this yourself.

```
bigList  = take (10^5) randomInts
hugeList = take (10^6) randomInts
```

We test this as follows

```
*Main> length bigList
100000
(0.02 secs, 4,522,952 bytes)
*Main> length(qsort bigList )
100000
(1.71 secs, 465,819,512 bytes)
*Main> length(bstsort bigList )
100000
(2.45 secs, 1,013,907,712 bytes)
*Main> length hugeList
1000000
(0.02 secs, 0 bytes)
*Main> length (qsort hugeList)
1000000
(21.94 secs, 7,578,153,768 bytes)
*Main> length (bstsort hugeList)
1000000
(37.34 secs, 12,514,195,728 bytes)
*Main> qsort hugeList == bstsort hugeList
True
(58.45 secs, 18,272,526,856 bytes)
*Main>
```

You may wish to perform your own tests and experiments with merge sort.

Other kinds of trees

Rose trees

A video discussing this section is available on Canvas.

Instead of having zero-branching (as `Empty`) and binary branching (as `Fork`) as in binary trees, we can branch any number of times, including zero, one, two, three, ..., depending on the length of the list in the following definition:

```
data Rose a = Branch a [Rose a]
```

Notice that there is no empty rose tree, but there is a rose tree with a label and no subtrees (or rather an empty list of subtrees). For example, the size of a rose tree can be defined as follows, so that it is always a positive number:

```
rsize :: Rose a -> Integer
rsize (Branch _ ts) = 1 + sum [rsize t | t <- ts]
```

This can be equivalently written as

```
rsize' :: Rose a -> Integer
rsize' (Branch _ ts) = 1 + sum (map rsize' ts)
```

The height is a bit trickier. The perhaps obvious definition

```
rheight :: Rose a -> Integer
rheight (Branch _ ts) = 1 + maximum [rheight t | t <- ts] -- wrong
```

doesn't work because the maximum of the empty list is not defined. Here is a correct definition:

```
rheight :: Rose a -> Integer
rheight (Branch _ []) = 0
rheight (Branch _ ts) = 1 + maximum [rheight t | t <- ts]
```

The second definition can be understood as follows. In an expression `Branch x ts`, we call `x` the root of the tree and the list `ts` of subtrees a `forest`. Then the height of the forest `ts` is `maximum (0 : [rheight' t | t <- ts])`. In particular, if `ts` is empty then its height 0, with the same convention as before. But the height of `Branch x ts` is 1 plus the height of the forest `ts`, accounting for the root node `x`.

> **Note:** The terminology "rose tree" is popular in functional programming. In mathematics and other areas of computer science, these kinds of trees are more commonly referred to as a "rooted planar trees" or "rooted ordered trees".

Game trees

A video discussing this section is available on Canvas.

Suppose we have a type of boards and a type of moves, where for any given board we know what are the possible moves and the board that results from playing each of those moves. Given an initial board, we can produce a game tree of all possible plays:

```
data GameTree board move = Node board [(move, GameTree board move)] deriving (Show)

gameTree :: (board -> [(move,board)]) -> board -> GameTree board move
gameTree plays board = Node board [(m, gameTree plays b) | (m,b) <- plays board]
```

To make this a bit more concrete, let's apply it to the game of Nim. In this case, the "board" is a collection of heaps of objects that we represent as a list of `Integer` s (since only the counts matter), and a move consists of picking a particular heap and removing some number of objects, represented by a pair of an `Int` (for the index of the heap) and an `Integer` (for the number of objects to remove):

```
type NimBoard = [Integer]
data NimMove = Remove Int Integer  deriving (Show,Eq)
```

The following "plays" function describes all the legal moves from a given Nim position:

```
nimPlays :: NimBoard -> [(NimMove,NimBoard)]
nimPlays heaps = [(Remove i k, (hs ++ h-k : hs'))
                 | i <- [0..length heaps-1],
                   let (hs, h:hs') = splitAt i heaps,
                   k <- [1..h]]
```

By passing this as the first argument to `gameTree`, we can compute the entire game tree from a given Nim position:

```
nim :: [Integer] -> GameTree NimBoard NimMove
nim = gameTree nimPlays
```

(Notice that this definition is "point-free" and that we've only partially applied the function `gameTree`. The above definition is completely equivalent to the definition `nim initHeaps = gameTree nimPlays initHeaps`, just more concise.)

Let's try it out with Nim:

```
> nim [2]
Node [2] [(Remove 0 1,Node [1] [(Remove 0 1,Node [0] [])]),(Remove 0 2,Node [0] [])]
> nim [2,1]
Node [2,1] [(Remove 0 1,Node [1,1] [(Remove 0 1,Node [0,1] [(Remove 1 1,Node [0,0] [])]),(Remove 1 1,Node [
> nim [1,1,1]
Node [1,1,1] [(Remove 0 1,Node [0,1,1] [(Remove 1 1,Node [0,0,1] [(Remove 2 1,Node [0,0,0] [])]),(Remove 2
```

Nim is usually played as a two-player game under "misère" rules where the first player who can't make a move wins, but can also be played as a "normal" two-player game where the first player who can't make a move loses. The question of whether or not the first player has a *winning strategy* from a given position can be entirely encapsulated into the logic of game trees.

```
isWinning, isLosing :: Bool -> GameTree board move -> Bool
isWinning isMisere (Node b mgs)
        | null mgs  = isMisere
        | otherwise = any (isLosing isMisere)  [g | (m,g) <- mgs]
isLosing  isMisere (Node b mgs)
        | null mgs  = not (isMisere)
        | otherwise = all (isWinning isMisere) [g | (m,g) <- mgs]
```

Let's try it out:

```
> isWinning True (nim [2])
True
> isWinning True (nim [2,1])
True
> isWinning True (nim [1,1,1])
False
> isWinning False (nim [1,1,1])
True
```

All this deserves further discussion, of course. For the purposes of these notes, we just wanted to illustrate yet-another-kind of tree that we may wish to define and manipulate in Haskell and in (functional or non-functional) programming more generally.

Permutation trees, list permutations, and paths in such trees (hard)

A video discussing this section is available on Canvas.

We now consider the type of list-branching trees, with labels on the edges rather than nodes, where a leaf is given by empty branching. What this means is that instead of having exactly two subtrees, as in binary trees, we have a (possibly empty) list of subtrees. If this list is empty, we have a leaf. The labels are at the edges, as in game trees, rather than at the nodes, as in the above binary trees:

```
data Tree a = EBranch [(a, Tree a)] deriving (Show)
```

Then `EBranch []`, of type `Tree a`, is a leaf. The list of all paths from the root to leafs is constructed as follows:

```
fullPaths :: Tree a -> [[a]]
fullPaths (EBranch []) = [[]]
fullPaths (EBranch forest) = [x:p | (x,t) <- forest, p <- fullPaths t]
```

A forest is an element of the type `[(a, Tree a)]`, namely a list of trees paired with elements. The list of all paths from the root to any node, including leafs:

```
paths :: Tree a -> [[a]]
paths (EBranch forest) =  [] : [x:p | (x,t) <- forest, p <- paths t]
```

We now construct the permutation tree of a list, where the full paths of the tree are the permutations of the given list:

```
permTree :: Eq a => [a] -> Tree a
permTree xs = EBranch [ (x, permTree(xs \\\ x)) | x <- xs]
  where
    (\\\) :: Eq a => [a] -> a -> [a]
    []     \\\ _    = undefined
    (x:xs) \\\ y
      | x == y      = xs
      | otherwise   = x : (xs \\\ y)
```

Using this, we can compute the list of all permutations of a given list:

```
permutations :: Eq a => [a] -> [[a]]
permutations = fullPaths . permTree
```

You know that `n! = 1 * 2 * 3 * ... * n`. This is the factorial of `n`, which counts the number of permutations of a list with `n` distinct elements. Hence one (inefficient!) way to compute the factorial function is

```
factorial n = length(permutations [1..n])
```

The function `removals` defined below has bad time complexity (quadratic?). The function `removals2` defined below is conceptually more complicated but runs in linear time. It needs the function that given a list produces the list of pairs where the first component is a removed element and the second component is the given list with that element removed. We have two versions (one intuitively clear, and the other less clear but much faster):

```
removals, removals2 :: [a] -> [(a,[a])]
removals [] = []
removals (x:xs) = (x,xs) : map (\(y,ys) -> (y,x:ys)) (removals xs)
```

Here is another representation of lists to make certain list operations faster (called difference lists).

```
type DList a = [a] -> [a]

removals' :: DList a -> [a] -> [(a,[a])]
removals' f [] = []
removals' f (x:xs) = (x, f xs) : removals' (f.(x:)) xs

removals2 = removals' (\xs -> xs)
```

With this, we can get permutation trees without relying on equality constraints:

```
permTree2 :: [a] -> Tree a
permTree2 xs = EBranch [(y, permTree2 ys) | (y,ys) <- removals2 xs]

permutations2 :: [a] -> [[a]]
permutations2 = fullPaths . permTree2
```

But, as discussed above, it is much less clear why this algorithm should be correct (meaning that it does do what we have claimed it to do).

- Self-learning: find out about computing the list of permutations of a given list without going via trees.
- Convince yourself that our way with trees corresponds to some of the proposed ways without trees.

Expression trees

A video discussing this section is available on Canvas.

Sometimes it is useful to work with a more specialised data type of trees tailored to a particular problem. For instance, many compilers parse strings into *expression trees* and then process them to generate code.

Here we define a simple data type of numerical expression trees, together with an *evaluation* function that processes expression trees to produce values.

```
data Expr a = Value a
            | FromInteger Integer
            | Negate (Expr a)
            | Abs (Expr a)
            | SigNum (Expr a)
            | Add (Expr a) (Expr a)
            | Mul (Expr a) (Expr a)
```

This mimics the `Num` class defined in the prelude:

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

We then define evaluation as follows:

```
eval :: Num a => Expr a -> a
eval (Value x)       = x
eval (FromInteger n) = fromInteger n
eval (Negate e)      = negate (eval e)
eval (Abs e)         = abs(eval e)
eval (SigNum e)      = signum(eval e)
eval (Add e e')      = eval e + eval e'
eval (Mul e e')      = eval e * eval e'
```

Before considering an example, this time let's define our own `show` function instead of asking Haskell to do this with the `deriving` mechanism. To a first approximation, the class `Show` is defined as follows in the prelude:

```
class Show a where
  show :: a -> String
```

(In full detail, the `Show` class has many more specified functions, but all of them can be derived from `show`.) We define our `show` function for expression trees like so:

```
instance Show a => Show(Expr a) where
  show (Value x)       = show x
  show (FromInteger n) = "fromInteger(" ++ show n ++ ")"
  show (Negate e)      = "negate(" ++ show e  ++ ")"
  show (Abs e)         = "abs(" ++ show e ++ ")"
  show (SigNum e)      = "signum(" ++ show e ++ ")"
  show (Add e e')      = "(" ++ show e ++ "+" ++ show e' ++ ")"
  show (Mul e e')      = "(" ++ show e ++ "*" ++ show e' ++ ")"
```

Examples:

```
*Main> eval (Mul (Value 3) (Add (Value 7) (Value 6)))
39
*Main> Mul (Value 3) (Add (Value 7) (Value 6))
(3*(7+6))
*Main> show (Mul (Value 3) (Add (Value 7) (Value 6)))
```

```
"(3*(7+6))"
```

Types with a single constructor

If a type has a single constructor with a single argument, it can be defined with `newtype`. See Section 8.3 of the book (page 95 of the printed version). However, there is a subtle, semantic difference with the corresponding declaration `data`, which is discussed in the haskell wiki.

If a data type has a single constructor and any number of arguments, it can be defined with *field labels*. Here is an example from A Gentle Introduction to Haskell, Version 98:

```
data Point = Pt Float Float

pointx, pointy :: Point -> Float
pointx (Pt x _) = x
pointy (Pt _ y) = y
```

Using field labels, this can be equivalently defined as

```
data Point = Pt {pointx, pointy :: Float}
```

You can read this as `Point` has two fields, one called `pointx` and the other called `pointy`, both of type `Float`. Then both `Pt 1 2` and `Pt {pointx=1, pointy=2}` are allowed as (equivalent) values of this type. You can also use this in pattern matching:

```
norm (Pt {pointx = x, pointy = y}) = sqrt (x*x+y*y).
```

There is also an *update syntax*: For example, if `p` is a point, then `p {pointx = 2}` is a new point with the field `pointx` replaced by `2` and the same `pointy` field.