CS 457/557 Functional Programming

Lecture 7
Trees
Trees

• Trees are important data structures in computer science.
• Trees have interesting properties:
  – They usually are finite, but statically unbounded in size.
  – They often contain other non-trivial types within.
  – They are often polymorphic.
  – They may have differing “branching factors”.
  – They may have different kinds of leaf and branching nodes.
• Lots of interesting things can be modeled as trees
  – lists (linear branching)
  – shapes (see text)
  – programming language syntax trees
• In a lazy language it is possible to have infinite trees.
Examples

data List a = Nil | MkList a (List a)
data Tree a = Leaf a | Branch (Tree a) (Tree a)
data IntegerTree = IntLeaf Integer
                | IntBranch IntegerTree IntegerTree
data SimpleTree = SLeaf
                | SBranch SimpleTree SimpleTree
data ITree a = ILeaf
                | IBranch a (ITree a) (ITree a)
data FancyTree a b = FLeaf a
                | FBranch b (FancyTree a b) (FancyTree a b)
Match up the Trees

- IntegerTree
- Tree
- SimpleTree
- List
- ITree
- FancyTree
Functions on Trees

• Transforming one kind of tree into another:

\[
\text{mapTree} :: (a \rightarrow b) \rightarrow \text{Tree } a \rightarrow \text{Tree } b
\]
\[
\text{mapTree } f \ (\text{Leaf } x) \quad = \quad \text{Leaf } (f \ x)
\]
\[
\text{mapTree } f \ (\text{Branch } t1 \ t2) \quad = \quad \text{Branch } (\text{mapTree } f \ t1) \quad (\text{mapTree } f \ t2)
\]

• Collecting the items in a tree:

\[
\text{fringe } :: \text{Tree } a \rightarrow [a]
\]
\[
\text{fringe } (\text{Leaf } x) \quad = \quad [x]
\]
\[
\text{fringe } (\text{Branch } t1 \ t2) \quad = \quad \text{fringe } t1 \quad +\quad \text{fringe } t2
\]

• What kind of information is lost using \textbf{fringe}?
More Functions on Trees

\[ \text{treeSize} \quad :: \quad \text{Tree } a \rightarrow \text{Integer} \]
\[ \text{treeSize} \ (\text{Leaf } x) \quad = \quad 1 \]
\[ \text{treeSize} \ (\text{Branch } t1 \ t2) \quad = \quad \text{treeSize} \ t1 \ + \ \text{treeSize} \ t2 \]

\[ \text{treeHeight} \quad :: \quad \text{Tree } a \rightarrow \text{Integer} \]
\[ \text{treeHeight} \ (\text{Leaf } x) \quad = \quad 0 \]
\[ \text{treeHeight} \ (\text{Branch } t1 \ t2) \quad = \quad 1 \ + \ \max \ (\text{treeHeight} \ t1) \]
\[ \phantom{=} \quad \ (\text{treeHeight} \ t2) \]
Binary Search Trees

- InternalTrees (values at internal nodes) in sorted order.
- Used for efficient implementation of sets, dictionaries, etc.
  - Logarithmic access, update in average case

```haskell
data ITree a
    = ILeaf
    | IBranch a (ITree a) (ITree a)

elemTree :: Ord a => a -> ITree a -> Bool
elemTree v ILeaf = False
elemTree v (IBranch w l r)
    | v == w    = True
    | v < w     = elemTree v l
    | v > w     = elemTree v r
```
Building Search Trees

insertTree :: Ord a => a -> ITree a -> ITree a
insertTree v ILeaf = IBranch v ILeaf ILeaf
insertTree v (IBranch w l r)
    | v <= w    = IBranch w (insertTree v l) r
    | v > w     = IBranch w l (insertTree v r)

listToTree xs = foldr insertTree ILeaf xs
s = listToTree [1,4,3,5,2,9,8]
== (IBranch 8 (IBranch 2 (IBranch 1 ILeaf
          ILeaf)
          (IBranch 5 (IBranch 3 ILeaf
                    (IBranch 4 ILeaf
                    ILeaf)))
          ILeaf))
          (IBranch 9 ILeaf
          ILeaf))
Deleting Elements

defleteTree:: Ord a => a -> ITree a -> ITree a
defleteTree v ILeaf = ILeaf
defleteTree v (IBranch w l r)
    | v == w    = glue l r
    | v < w     = IBranch w (deleteTree v l) r
    | v > w     = IBranch w l (deleteTree v r)

glue:: ITree a -> ITree a -> ITree a
glue ILeaf r = r
glue l r = IBranch big l' r
    where (big,l') = largest l

largest:: ITree a -> (a,ITree a) -- largest elem, rest
largest (IBranch w l ILeaf) = (w,l)
largest (IBranch w l r) = (big,IBranch w l r')
    where (big,r') = largest r
Arithmetic Expressions

data Expr = C Float
   | Add Expr2 Expr2
   | Sub Expr2 Expr2
   | Mul Expr2 Expr2
   | Div Expr2 Expr2

Or, using infix constructor names:

data Expr = C Float
   | Expr :+: Expr
   | Expr :-: Expr
   | Expr :*: Expr
   | Expr :/: Expr

Infix constructors begin with a colon (:) , whereas ordinary constructor functions begin with an upper-case character.
Example

\[ e_1 = (C\ 10\ :+\ (C\ 8\ :/\ C\ 2))\ :*\ (C\ 7\ :-\ C\ 4) \]

\[ \text{evaluate} \quad :: \quad \text{Expr} \rightarrow \text{Float} \]
\[ \text{evaluate} \ (C\ x) \quad = \quad x \]
\[ \text{evaluate} \ (e_1\ :+\ e_2) \quad = \quad \text{evaluate} \ e_1\ +\ \text{evaluate} \ e_2 \]
\[ \text{evaluate} \ (e_1\ :-\ e_2) \quad = \quad \text{evaluate} \ e_1\ -\ \text{evaluate} \ e_2 \]
\[ \text{evaluate} \ (e_1\ :*\ e_2) \quad = \quad \text{evaluate} \ e_1\ *\ \text{evaluate} \ e_2 \]
\[ \text{evaluate} \ (e_1\ :/\ e_2) \quad = \quad \text{evaluate} \ e_1\ /\ \text{evaluate} \ e_2 \]

Main> evaluate e1
42.0