Advanced Programming
Handout 5

Recursive Data Types
(SOE Chapter 7)
Trees

- Trees are important data structures in computer science.
- Trees have interesting properties:
  - They are usually finite, but potentially unbounded in size.
  - They often contain other types (ints, strings, lists) within.
  - They are often polymorphic.
  - They may have differing “branching factors”.
  - They may have different kinds of leaf and branching nodes.
- Lots of interesting data structures are tree-like:
  - lists (linear branching)
  - arithmetic expressions (see SOE)
  - parse trees (for programming or natural languages)
  - etc., etc.
- In a lazy language like Haskell, we can even build infinite trees!
Examples

Note that this type declaration is recursive: **List** is mentioned on its right-hand side.

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 InternalTree a = ILeaf |
IBranch a (InternalTree a) (InternalTree a)
data FancyTree a b = FLeaf a |
FBranch b (FancyTree a b) (FancyTree a b)
Match up the Trees

- **IntegerTree**
  ```
  1
  'a'
  2
  'b' 'c'
  ```

- **Tree**
  ```
  'a'
  'b' 'c'
  ```

- **SimpleTree**
  ```
  'a'
  2
  'b' 'c'
  ```

- **List**

- **InternalTree**

- **FancyTree**
  ```
  6 9
  2
  'a'
  'b'
  ```
Functions on Trees

- Transforming a tree of \( a \)s into a tree of \( b \)s:

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

  \[
  \text{mapTree} \ f \ (\text{Leaf} \ x) = \text{Leaf} \ (f \ x)
  \]

  \[
  \text{mapTree} \ f \ (\text{Branch} \ t1 \ t2) = \text{Branch} \ (\text{mapTree} \ f \ t1) \ (\text{mapTree} \ f \ t2)
  \]

- Collecting the items in a tree:

  \[
  \text{fringe} :: \text{Tree} \ a \rightarrow \ [a]
  \]

  \[
  \text{fringe} \ (\text{Leaf} \ x) = [x]
  \]

  \[
  \text{fringe} \ (\text{Branch} \ t1 \ t2) = \text{fringe} \ t1 \ ++ \ \text{fringe} \ t2
  \]
More Functions on Trees

treeSize :: Tree a -> Integer
  treeSize (Leaf x)       = 1
  treeSize (Branch t1 t2) = treeSize t1 + treeSize t2


treeHeight :: Tree a -> Integer
  treeHeight (Leaf x)       = 0
  treeHeight (Branch t1 t2) = 1 + max (treeHeight t1) (treeHeight t2)
Capturing a Pattern of Recursion

Many of our functions on trees have similar structure. Can we apply the abstraction principle?

Of course we can!

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

\[
\text{foldTree } \text{combineFn } \text{leafFn } (\text{Leaf } x) =
\]

\[
\text{leafFn } x
\]

\[
\text{foldTree } \text{combineFn } \text{leafFn } (\text{Branch } t1 \ t2) =
\]

\[
\text{combineFn } (\text{foldTree } \text{combineFn } \text{leafFn } t1)
\]

\[
(\text{foldTree } \text{combineFn } \text{leafFn } t2)
\]
Using foldTree

With \texttt{foldTree} we can redefine the previous functions like this:

\begin{align*}
\text{mapTree } f & = \text{foldTree } \text{Branch } \text{fun} \\
& \quad \text{where fun } x = \text{Leaf } (f \ x) \\
\text{fringe} & = \text{foldTree } (++) \text{fun} \\
& \quad \text{where fun } x = [x] \\
\text{treeSize} & = \text{foldTree } (+) (\text{const } 1) \\
& \quad \text{where const } x \ y = x \\
\text{treeHeight} & = \text{foldTree } \text{fun} (\text{const } 0) \\
& \quad \text{where const } x \ y = x \\
& \quad \quad \text{fun } x \ y = 1 + \text{max } x \ y
\end{align*}

Partial application again!
Arithmetic Expressions

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

Or, using infix constructor names:

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

e1 = (C 10 :+ (C 8 :/ C 2)) :* (C 7 :- C 4)

evaluate :: Expr -> Float
evaluate (C x) = x
evaluate (e1 :+: e2) = evaluate e1 + evaluate e2
evaluate (e1 ::- e2) = evaluate e1 - evaluate e2
evaluate (e1 :*: e2) = evaluate e1 * evaluate e2
evaluate (e1 :/: e2) = evaluate e1 / evaluate e2

Main> evaluate e1
42.0
Chapter 8

A Module of Regions
The Region Data Type

- A region represents an area on the two-dimensional Cartesian plane.
- It is represented by a tree-like data structure.

```haskell
data Region =
    Shape Shape               -- primitive shape
  | Translate Vector Region   -- translated region
  | Scale Vector Region       -- scaled region
  | Complement Region         -- inverse of region
  | Region `Union` Region     -- union of regions
  | Region `Intersect` Region -- intersection of regions
  | Empty

type Vector = (Float, Float)
```
Questions about Regions

- What is the strategy for writing functions over regions?
- Is there a fold-function for regions?
  - How many parameters does it have?
  - What is its type?
- Can one define infinite regions?
- *What does a region mean?*
Sets and Characteristic Functions

- How can we represent an infinite set in Haskell? E.g.:
  - the set of all even numbers
  - the set of all prime numbers

- We could use an infinite list, but then searching it might take a very long time! (Membership becomes semi-decidable.)

- The *characteristic function* for a set containing elements of type $z$ is a function of type $z \rightarrow \text{Bool}$ that indicates whether or not a given element is in the set. Since that information completely characterizes a set, we can use it to represent a set:

  ```haskell
  type Set a = a -> Bool
  ```

- For example:

  ```haskell
  even :: Set Integer       -- i.e., Integer -> Bool
  even x = (x `mod` 2) == 0
  ```
Combining Sets

- If sets are represented by characteristic functions, then how do we represent the:
  - union of two sets?
  - intersection of two sets?
  - complement of a set?

- In-class exercise – define the following Haskell functions:

```haskell
union s1 s2 =
intersect s1 s2 =
complement s =
```

- We will use these later to define similar operations on regions.
The “meaning” (or “denotation”) of a region can be expressed as its characteristic function -- i.e.,

*a region denotes the set of points contained within it.*
Characteristic Functions for Regions

- We define the meaning of regions by a function:
  
  ```
  containsR :: Region -> Coordinate -> Bool
  type Coordinate = (Float, Float)
  ```

- Note that `containsR r :: Coordinate -> Bool`, which is a characteristic function. So `containsR` “gives meaning to” regions.

- Another way to see this:
  
  ```
  containsR :: Region -> Set Coordinate
  ```

- We can define `containsR` recursively, using pattern matching over the structure of a `Region`.

- Since the base cases of the recursion are primitive shapes, we also need a function that gives meaning to primitive shapes; we will call this function `containsS`. 
Rectangle

\[
\text{Rectangle } s_1 \ s_2 \ \text{`containsS`} \ (x,\ y) = \begin{align*}
&= \text{let } t_1 = s_1/2 \\
&\quad t_2 = s_2/2 \\
&\quad \text{in } -t_1 \leq x \ \&\ x \leq t_1 \ \&\ -t_2 \leq y \ \&\ y \leq t_2
\end{align*}
\]
Ellipse

Ellipse $r_1 \ r_2$ `containsS` $(x,y)$

$= (x/r_1)^2 + (y/r_2)^2 \leq 1$
The Left Side of a Line

For a ray directed from point \(a\) to point \(b\), a point \(p\) is to the left of the ray (facing from \(a\) to \(b\)) when:

\[
isLeftOf :: \text{Coordinate} \to \text{Ray} \to \text{Bool}
\]

\[
(p_x, p_y) \ `\text{isLeftOf} ` ((a_x, a_y), (b_x, b_y))
\]

\[
= \text{let } (s, t) = (p_x-a_x, p_y-a_y) \\
(u, v) = (p_x-b_x, p_y-b_y) \\
in \ s \ast v \geq t \ast u
\]

type Ray = (\text{Coordinate}, \text{Coordinate})
A point $p$ is contained within a (convex) polygon if it is to the left of every side, when they are followed in counter-clockwise order.

```haskell
Polygon pts `containsS` p
  = let shiftpts = tail pts ++ [head pts]
      leftOfList = map isLeftOfp (zip pts shiftpts)
      isLeftOfp p' = isLeftOf p p'
      in and leftOfList
```
Right Triangle

\[
\text{RtTriangle } s_1 \text{ } s_2 \ `\text{containsS} \text{' } p \\
= \text{Polygon } [(0,0),(s_1,0),(0,s_2)] \ `\text{containsS} \text{' } p
\]
Putting it all Together

containsS :: Shape -> Vertex -> Bool
Rectangle s1 s2 `containsS` (x,y)
    = let t1 = s1/2; t2 = s2/2
        in -t1<=x && x<=t1 && -t2<=y && y<=t2
Ellipse r1 r2 `containsS` (x,y)
    = (x/r1)^2 + (y/r2)^2 <= 1
Polygon pts `containsS` p
    = let shiftpts = tail pts ++ [head pts]
        leftOfList = map isLeftOfp (zip pts shiftpts)
        isLeftOfp p' = isLeftOf p p'
        in and leftOfList
RtTriangle s1 s2 `containsS` p
    = Polygon [(0,0),(s1,0),(0,s2)] `containsS` p
Defining \texttt{containsR}

\begin{align*}
\text{containsR} &: \text{Region} \rightarrow \text{Vertex} \rightarrow \text{Bool} \\
\text{Shape } s \ `\text{containsR}` p &= s \ `\text{containsS}` p \\
\text{Translate } (u,v) \ r \ `\text{containsR}` (x,y) &= r \ `\text{containsR}` (x-u,y-v) \\
\text{Scale } (u,v) \ r \ `\text{containsR}` (x,y) &= r \ `\text{containsR}` (x/u,y/v) \\
\text{Complement } r \ `\text{containsR}` p &= \text{not} \ (r \ `\text{containsR}` p) \\
\text{r1 `Union` r2} \ `\text{containsR}` p &= \text{r1 `containsR` p} \ || \ \text{r2 `containsR` p} \\
\text{r1 `Intersect` r2} \ `\text{containsR}` p &= \text{r1 `containsR` p} \ &\& \ \text{r2 `containsR` p} \\
\text{Empty} \ `\text{containsR}` p &= \text{False}
\end{align*}
Applying the Semantics

Having defined the meanings of regions, what can we use them for?

- In Chapter 10, we will use the `containsR` function to test whether a mouse click falls within a region.
- We can also use the interpretation of regions as characteristic functions to reason about abstract properties of regions. E.g., we can show (by calculation) that `Union` is commutative, in the sense that:
  
  \[(r_1 \ `\text{Union}` \ r_2) \ `\text{containsR}` \ p \Rightarrow (r_2 \ `\text{Union}` \ r_1) \ `\text{containsR}` \ p\]

  (and vice versa)

This is very cool: Instead of having a separate “program logic” for reasoning about properties of programs, we can prove many interesting properties directly by calculation on Haskell program texts.

Unfortunately, we will not have time to pursue this topic further in this class.