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!

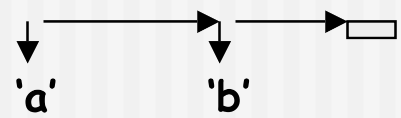
Examples

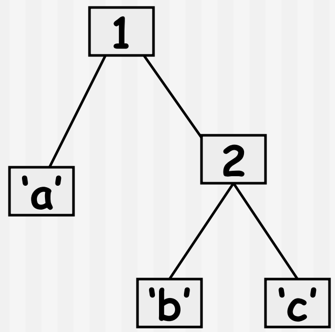
Note that this type declaration is *recursive*: `List` is mentioned on its right-hand side

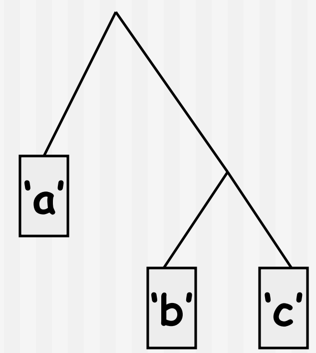
```
data List a           = Nil
                      | MkList a (List a)
data Tree a           = Leaf a
                      | Branch (Tree a) (Tree a)
data IntegerTree      = IntLeaf Integer
                      | IntBranch IntegerTree IntegerTree
data SimpleTree       = SLeaf
                      | 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)
```

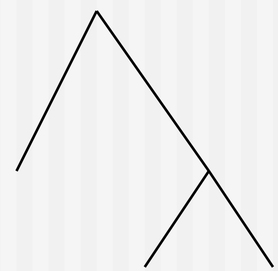
Match up the Trees

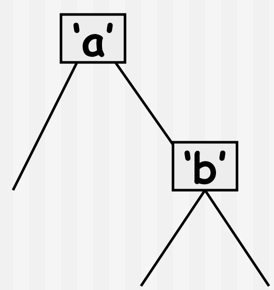
- `IntegerTree`
- `Tree`
- `SimpleTree`
- `List`
- `InternalTree`
- `FancyTree`

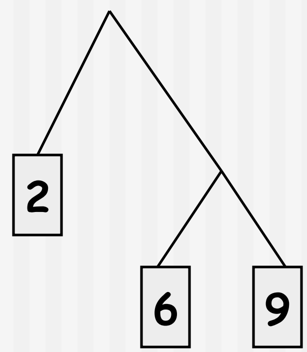












Functions on Trees

- Transforming a tree of `a`s into a tree of `b`s :

```
mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf x)        = Leaf (f x)
mapTree f (Branch t1 t2) = Branch (mapTree f t1)
                                  (mapTree f t2)
```

- Collecting the items in a tree:

```
fringe               :: Tree a -> [a]
fringe (Leaf x)       = [x]
fringe (Branch t1 t2) = fringe t1 ++ 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!

```
foldTree :: (a -> a -> a) -> (b -> a) -> Tree b -> a
foldTree combineFn leafFn (Leaf x) =
          leafFn x
foldTree combineFn leafFn (Branch t1 t2) =
          combineFn (foldTree combineFn leafFn t1)
                    (foldTree combineFn leafFn t2)
```

Using foldTree

With **foldTree** we can redefine the previous functions like this:

```
mapTree f   = foldTree Branch fun
              where fun x = Leaf (f x)

fringe      = foldTree (++) fun
              where fun x = [x]
treeSize    = foldTree (+) (const 1)
              where const x y = x
treeHeight = foldTree fun (const 0)
              where const x y = x
                    fun x y = 1 + max x y
```

Partial application again!

Arithmetic Expressons

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

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

```
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.

```
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 -> 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:

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

- For example:

```
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:

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

- We will use these later to define similar operations on regions.

Semantics of Regions

The “meaning” (or “denotation”) of a region can be expressed as its characteristic function -- i.e.,

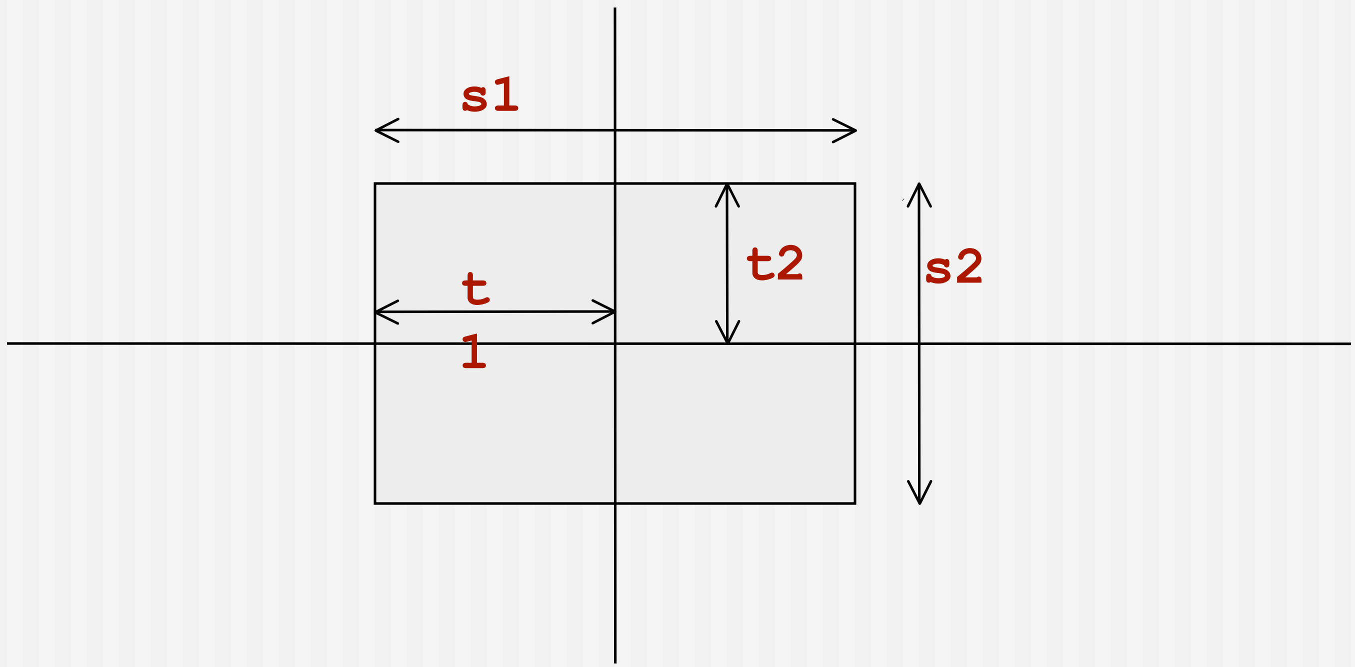
a region denotes the set of points contained within it.

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`.

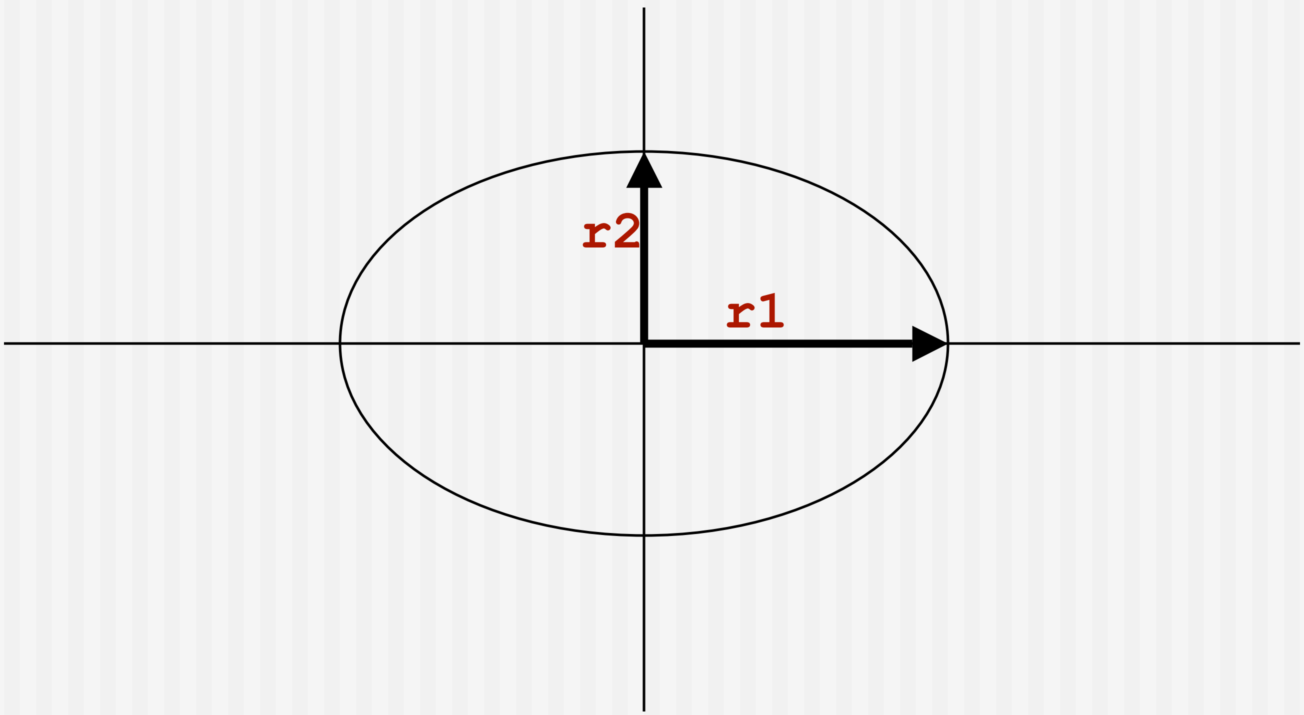
Rectangle

```
Rectangle s1 s2 `containsS` (x,y)
  = let t1 = s1/2
        t2 = s2/2
    in -t1<=x && x<=t1 && -t2<=y && y<=t2
```



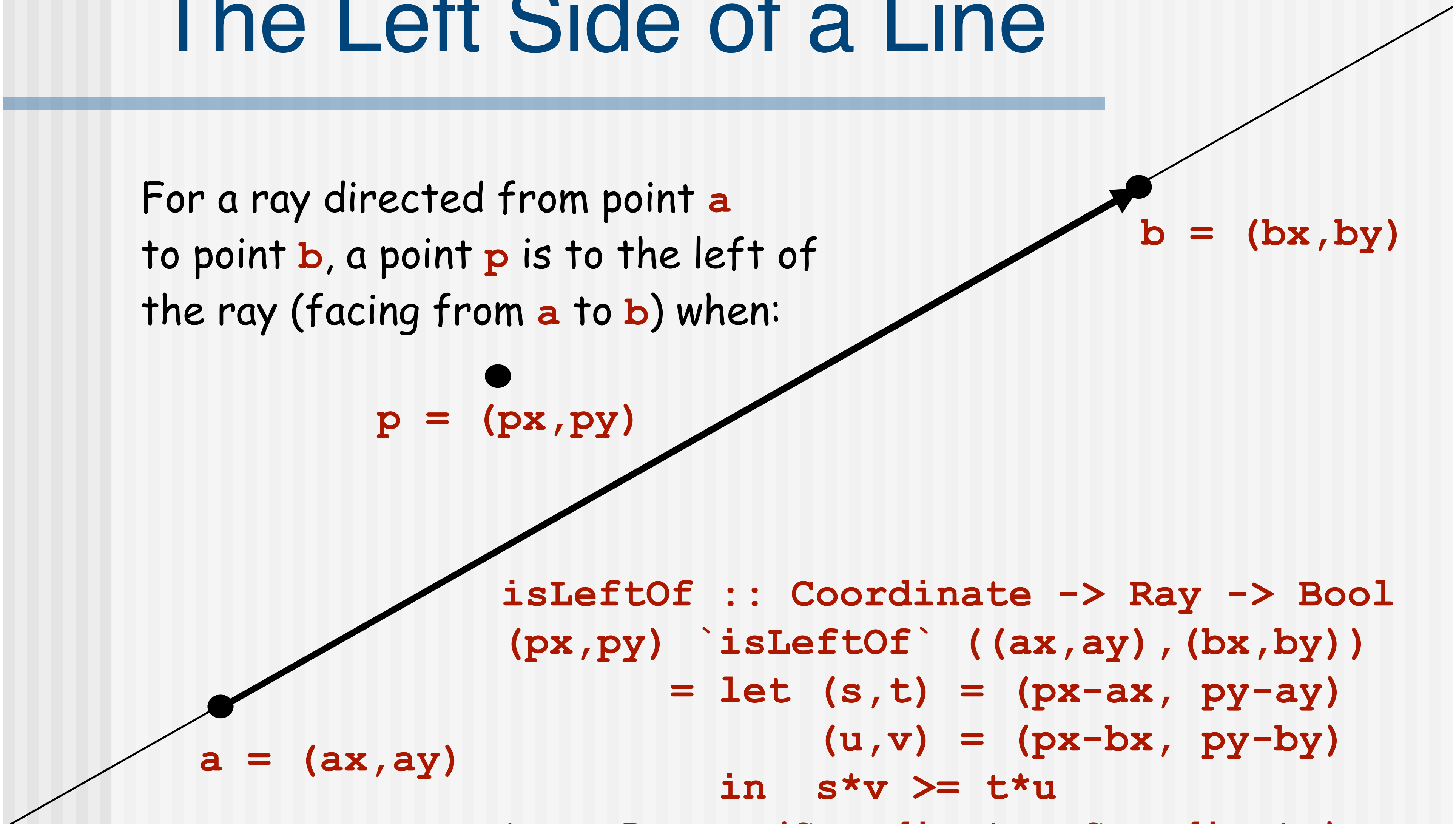
Ellipse

```
Ellipse r1 r2 `containsS` (x,y)
      = (x/r1)^2 + (y/r2)^2 <= 1
```



The Left Side of a Line

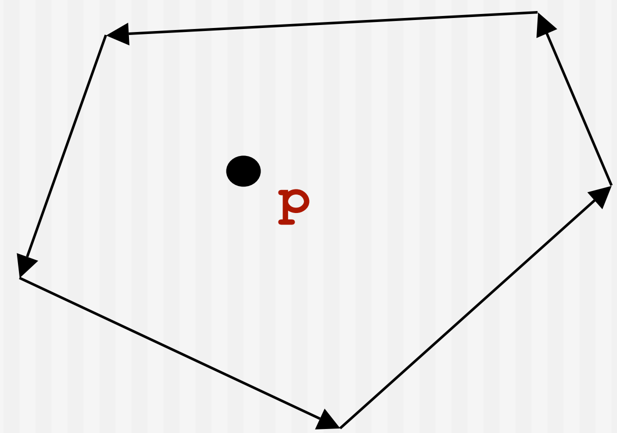
For a ray directed from point **a** to point **b**, a point **p** is to the left of the ray (facing from **a** to **b**) when:




```
isLeftOf :: Coordinate -> Ray -> Bool
(px,py) `isLeftOf` ((ax,ay),(bx,by))
        = let (s,t) = (px-ax, py-ay)
              (u,v) = (px-bx, py-by)
          in  s*v >= t*u
type Ray = (Coordinate, Coordinate)
```

Polygon

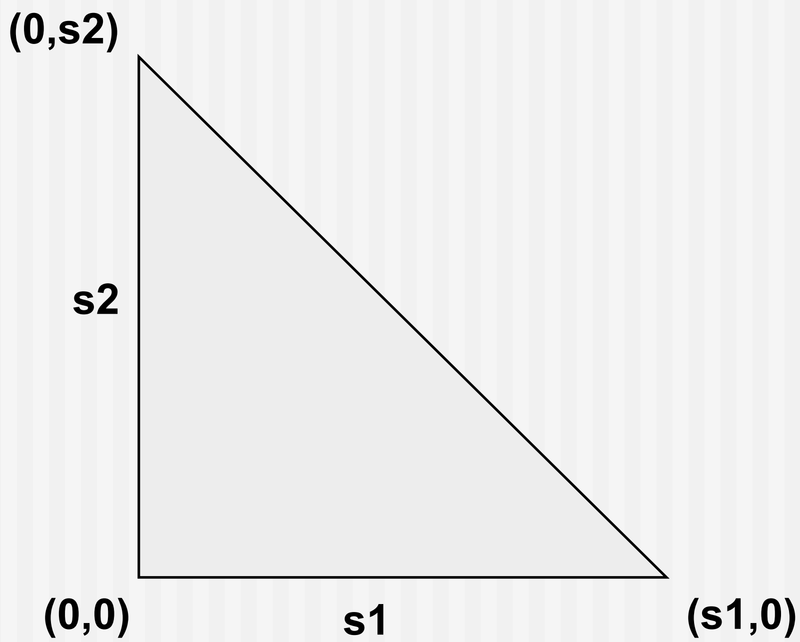
A point **p** is contained within a (convex) polygon if it is to the left of every side, when they are followed in counter-clockwise order.




```
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

```
RtTriangle s1 s2 `containsS` p
    = Polygon [(0,0),(s1,0),(0,s2)] `containsS` 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 `containsR`

```
containsR :: Region -> Vertex -> Bool
Shape s `containsR` p = s `containsS` p
Translate (u,v) r `containsR` (x,y)
                      = r `containsR` (x-u,y-v)
Scale (u,v) r     `containsR` (x,y)
                      = r `containsR` (x/u,y/v)
Complement r      `containsR` p
                      = not (r `containsR` p)
r1 `Union` r2     `containsR` p
          = r1 `containsR` p || r2 `containsR` p
r1 `Intersect` r2 `containsR` p
          = r1 `containsR` p && r2 `containsR` p
Empty     `containsR` p = False
```

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:

  for any regions **`r1`** and **`r2`** and any vertex **`p`** ,

  ```
    (r1 `Union` r2) `containsR` p
  ➔ (r2 `Union` r1) `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.