CS 457/557 Functional Programming

Lecture 8

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
   deriving Show

type Vector = (Float, Float)
```

Questions about Regions

- Why is `Region` tree-like?
- 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          -- 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:

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

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

Why Regions?

Regions (as defined in the text) are interesting because:

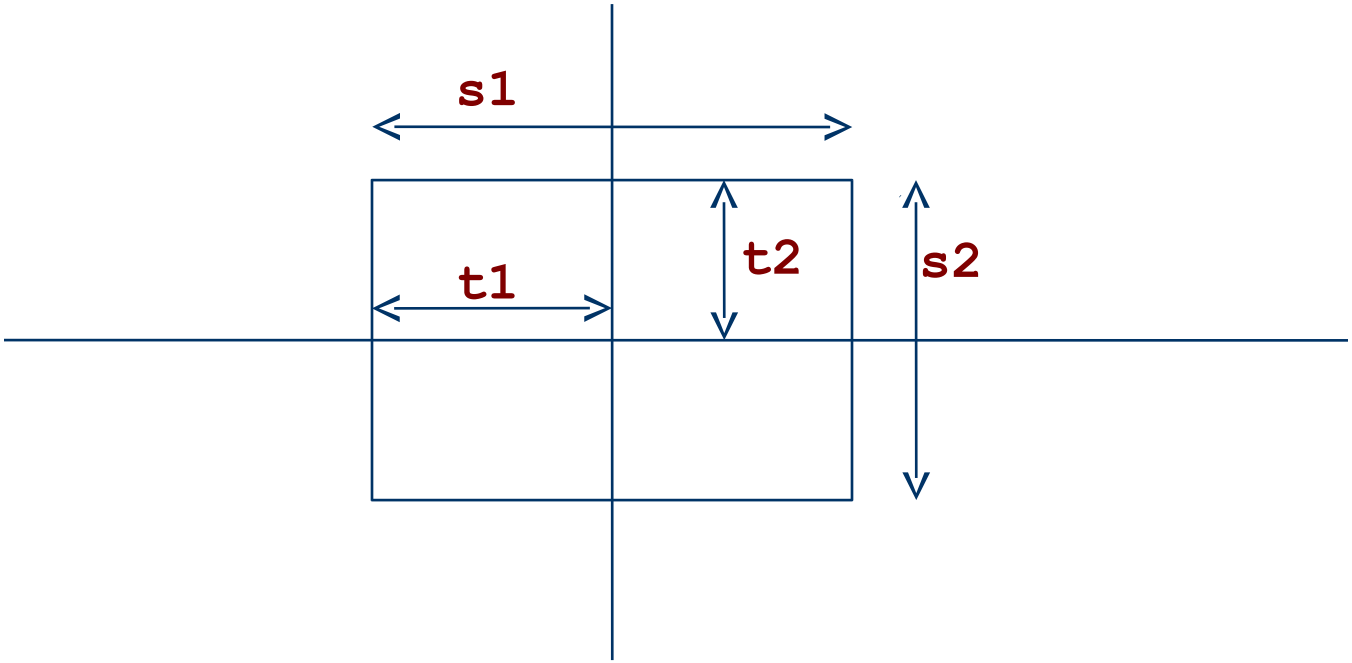
- They allow us to build complex “shapes” from simpler ones.
- They illustrate the use of tree-like data structures.
- They “solve” the problem of having rectangles and ellipses centered about the origin.
- Their meaning can be given as characteristic functions, since *a region denotes the set of points contained within it.*

Characteristic Functions for Regions

- We define the meaning of regions by a function:
  `containsR :: Region -> Coordinate -> Bool`
- Here `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`.

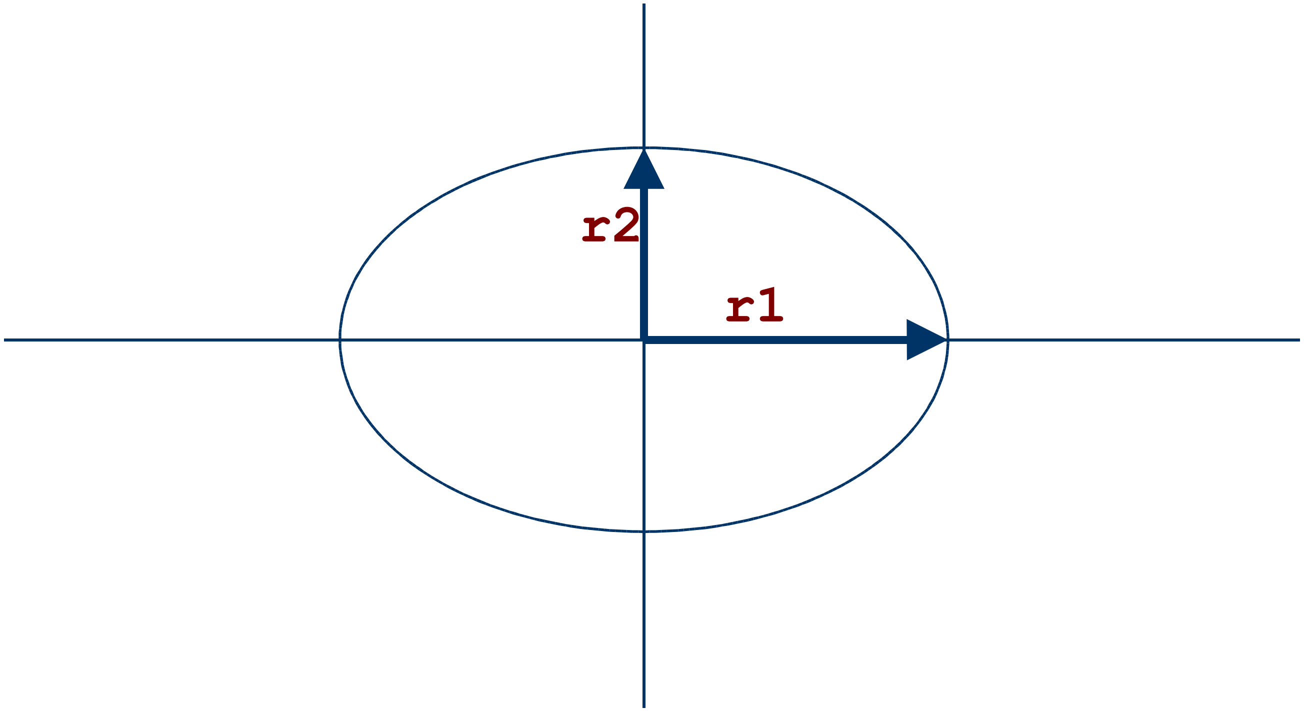
Rectangle

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



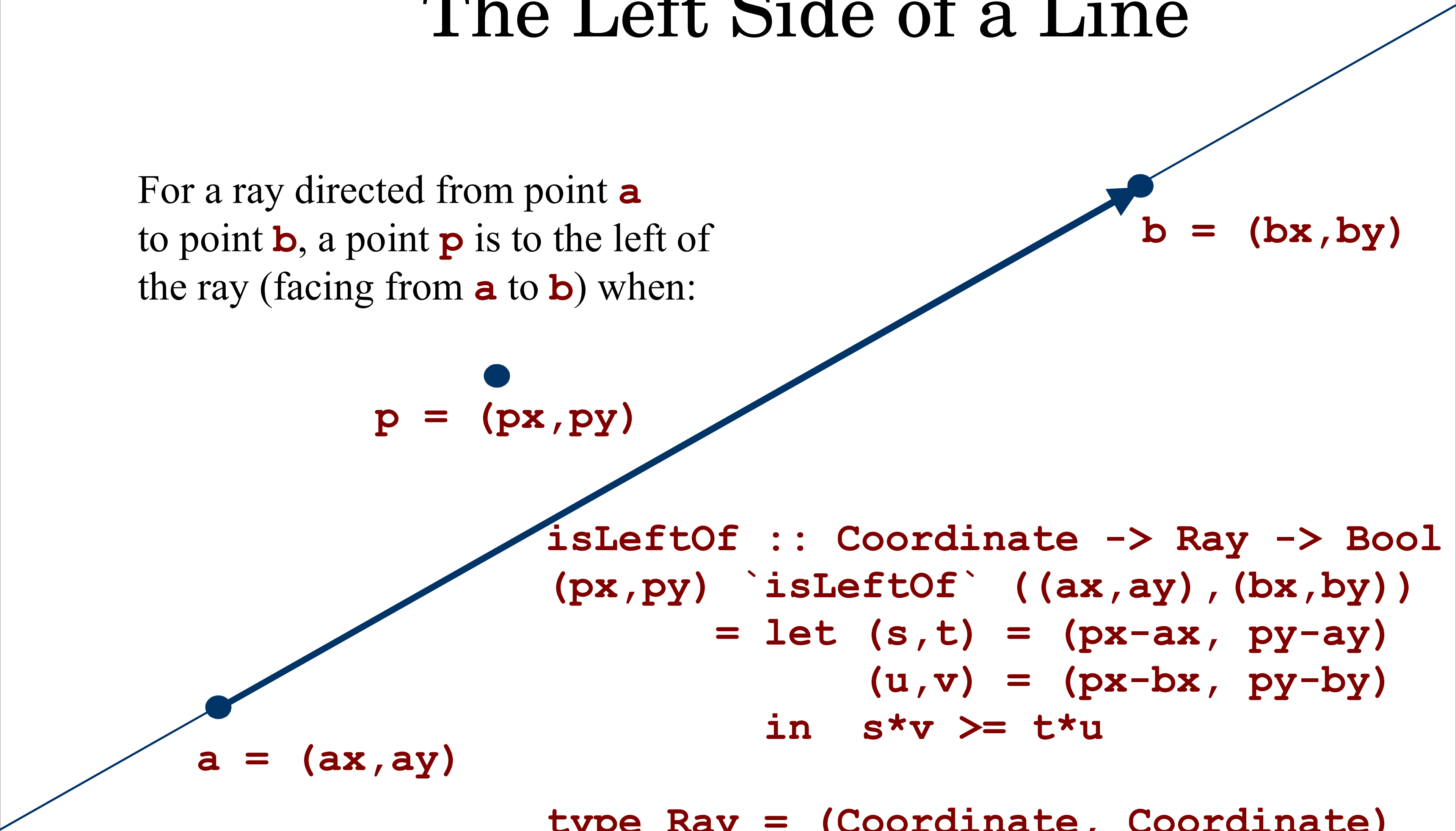
Ellipse

```
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:

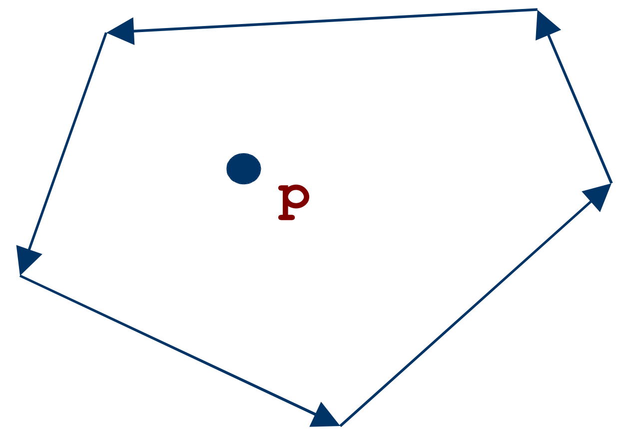



```
isLeftOf :: Coordinate -> Ray -> Bool
(px,py) `isLeftOf` ((ax,ay),(bx,by))
          = let (s,t) = (px-ax, py-ay)
                (u,v) = (px-bx, py-by)
            in  s*v >= t*u
```

```
type Ray = (Coordinate, Coordinate)
```

Polygon

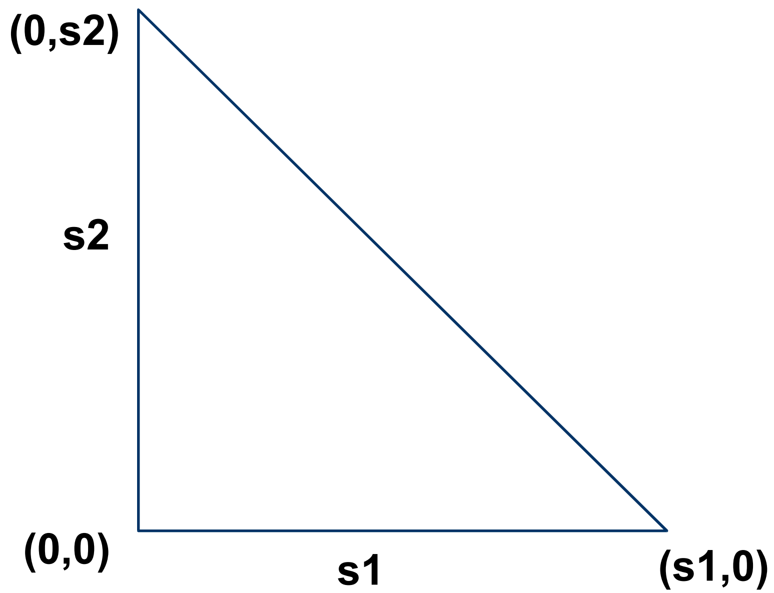
A point **p** is contained within a (convex) polygon if it is to the left of every side, when they are followed in counter-clockwise order.




```
Polygon pts `containsS` p
  = let shiftpts = tail pts ++ [head pts]
        leftOfList = map (isLeftOfp p)
                         (zip pts shiftpts)
    in foldr (&&) True leftOfList
```

Right Triangle

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



Putting it all Together

```
containsS :: Shape -> Coordinate -> 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 p) (zip pts shiftpts)
     in foldr (&&) True leftOfList
RtTriangle s1 s2 `containsS` p
   = Polygon [(0,0),(s1,0),(0,s2)] `containsS` p
```

Defining **containsR** using Recursion

```
containsR :: Region -> Coordinate -> 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
```

An Algebra of Regions

- Note that, for any **`r1`**, **`r2`**, and **`r3`**:

```
(r1 `Union` (r2 `Union` r3)) `containsR` p
```

  if and only if:

```
(r1 `Union` r2) `Union` r3)) `containsR` p
```

  which we can abbreviate as:

```
(r1 `Union` (r2 `Union` r3))
        ≡ ((r1 `Union` r2) `Union` r3)
```

- In other words, **`Union`** is *associative*.

- We can prove this fact via calculation.

Proof of Associativity

```
(r1 `Union` (r2 `Union` r3)) `containsR` p
= (r1 `containsR` p) ||
         ((r2 `Union` r3) `containsR` p)
= (r1 `containsR` p) ||
         ((r2 `containsR` p) || (r3 `containsR` p))
= ((r1 `containsR` p) || (r2 `containsR` p)) ||
         (r3 `containsR` p)
= ((r1 `Union` r2) `containsR` p) ||
         (r3 `containsR` p)
= ((r1 `Union` r2) `Union` r3) `containsR` p
```

(Note that the proof depends on the associativity of **(||)**, which can also be proved by calculation, but we take as given.)

More Axioms

There are many useful axioms for regions:

1) `Union` and `Intersect` are *associative*.
2) `Union` and `Intersect` are *commutative*.
3) `Union` and `Intersect` are *distributive*.
4) `Empty` and `univ = Complement Empty` are *zeros* for `Union` and `Intersect`, respectively.
5) `` r `Union` Complement r `` ≡ `univ` and
   `` r `Intersect` Complement r `` ≡ `Empty`

This set of axioms captures what is called a *boolean algebra*.