Preparing for FRP

Shapes, Regions, and Drawing
Shape types from the Text

data Shape = Rectangle Float Float
           | Ellipse Float Float
           | RtTriangle Float Float
           | Polygon [(Float,Float)]

  deriving Show

• Deriving Show
  – tells the system to build a `show` function for the type `Shape`

• Using Shape - Functions returning shape objects

  circle radius = Ellipse radius radius radius
  square side = Rectangle side side
Functions over Shape

- Functions over shape can be defined using pattern matching

```haskell
area :: Shape -> Float

area (Rectangle s1 s2)  = s1 * s2
area (Ellipse r1 r2)    = pi * r1 * r2
area (RtTriangle s1 s2) = (s1 *s2) / 2
area (Polygon (v1:pts)) = polyArea pts

where polyArea :: [ (Float,Float) ] -> Float
  polyArea (v2 : v3 : vs) = triArea v1 v2 v3 +
                           polyArea (v3:vs)
  polyArea _    = 0
```

Note use of prototype
Note use of nested patterns
Note use of wild card pattern (matches anything)
Poly = [A, B, C, D, E, F]

Area = Area(Triangle [A, B, C]) + Area(Poly[A, C, D, E, F])
TriArea

triArea v1 v2 v3 =
  let a = distBetween v1 v2
  b = distBetween v2 v3
  c = distBetween v3 v1
  s = 0.5*(a+b+c)
in sqrt (s*(s-a)*(s-b)*(s-c))

distBetween (x1,y1) (x2,y2)
  = sqrt (((x1-x2)^2 + (y1-y2)^2)
Interacting with the world through graphics

• Our first example of an action is found in chapter 3
• The action is to pop up a window and to draw pictures in the window.
module Main where
import SOE

ex0 =
  runGraphics(
    do { w <- openWindow "First window" (300,300)
     ; drawInWindow w (text (100,200) "hello world")
     ; k <- getKey w
     ; closeWindow w
     } )

This imports a library, SOE, it contains many functions.
Graphics Operators

• `openWindow :: String -> (Int,Int) -> IO Window`
  – opens a titled window of a particular size

• `drawInWindow :: Window -> Graphic -> IO ()`
  – Takes a window and a `Graphic` object and draws it
  – Note the return type of `IO()`

• `getKey :: Window -> IO Char`
  – Waits until any key is pressed and then returns that character

• `closeWindow :: Window -> IO ()`
  – closes the window

• try it out
A Bug in the code?

getKey :: Window -> IO Char
getKey win = do
  ch <- getKeyEx win True
  if (ch == '\x0') then return ch
  else getKeyEx win False

getKey :: Window -> IO Char
getKey win = do
  ch <- getKeyEx win True
  if not(ch == '\x0') then return ch
  else getKeyEx win False
spaceClose :: Window -> IO ()
spaceClose w =
  do { k <- getKey w
       ; if k == ' ' then closeWindow w
         else spaceClose w
  }

ex1 =
  runGraphics(
    do { w <- openWindow "Second Program" (300,300)
         ; drawInWindow w (text (100,200) "hello Again")
         ; spaceClose w
    })
Drawing Primitive Shapes

• The Graphics libraries contain primitives for drawing a few primitive shapes.
• We will build complicated drawing programs from these primitives

```haskell
ellipse :: Point -> Point -> Graphic
shearEllipse :: Point -> Point -> Point -> Graphic
line :: Point -> Point -> Graphic
polygon :: [Point] -> Graphic
polyline :: [Point] -> Graphic
```
Coordinate Systems

(0, 0)

Increasing x-axis

Increasing y-axis
Example Program

ex2 =
    runGraphics(
        do { w <- openWindow "Draw some shapes" (300,300)
          ; drawInWindow w (ellipse (0,0) (50,50))
          ; drawInWindow w
            (shearEllipse (0,60) (100,120) (150,200))
          ; drawInWindow w
            (withColor Red (line (200,200) (299,275)))
          ; drawInWindow w
            (polygon [(100,100),(150,100),(160,200)])
          ; drawInWindow w
            (withColor Green
              (polyline [(100,200),(150,200),
                         (160,299),(100,200)]))
          ; spaceClose w
        } )
; drawInWindow w
  (ellipse (0,0) (50,50))

; drawInWindow w
  (shearEllipse (0,60)
   (100,120)
   (150,200))

; drawInWindow w
  (withColor Red
   (line (200,200)
     (299,275)))

; drawInWindow w
  (polygon [(100,100),
            (150,100),
            (160,200)])

; drawInWindow w
  (withColor Green
   (polyline
    [(100,200),(150,200),
     (160,299),(100,200)])
})
Building Programs

• We’d like to build bigger things from these small pieces
• Perhaps things such as fractals
  – Example: Sierpinski’s Triangle
    a repeated drawing of a triangle at repeatedly smaller sizes.

• Key Idea
  Separate pure computation from action
Remember \( y \) increases as we go down the page.

\[
\text{size} \times \text{size} + \text{size} \times \text{size} = \text{hyp} \times \text{hyp}
\]
fillTri x y size w =
  drawInWindow w
  (withColor Blue
   (polygon [(x,y),
             (x+size,y),
             (x,y-size)]))

minSize = 8
Sierpinski’s Triangle

sierpinskiTri w x y size =
  if size <= minSize
    then fillTri x y size w
  else let size2 = size `div` 2
    in do { sierpinskiTri w x y size2
           ; sierpinskiTri w x (y-size2) size2
           ; sierpinskiTri w (x + size2) y size2
         }

ex3 =
  runGraphics(
    do { w <- openWindow "Sierpinski's Tri" (400,400)
         ; sierpinskiTri w 50 300 256
         ; spaceClose w
       } )
Question?

- What’s the largest triangle \texttt{sierpinskitri} ever drawn?
- How come the big triangle is drawn?
Abstraction

- Drawing a polygon in a particular window, with a particular color is a pretty common thing. Let's give it a name.

```haskell
drawPoly w color points =
  drawInWindow w
  (withColor color (polygon points))
```
Draw a snowflake
Geometry of Snow flakes

- Snow flakes are six sided
- They have repeated patterns
- An easy six sided figure is the Star of David
  - Constructed from two equilateral triangles

![Diagram of a circle and a triangle with labels for radius, side, and height.]

Radius = 2/3 * height
height = sqrt(side*side - (side*side)/4)
2 triangles with common center
Compute the corners

eqTri side (x,y) =
  let xf = fromIntegral x
  yf = fromIntegral y
  sideDiv2 = side / 2.0
  height = sqrt( side*side -
                 (sideDiv2 * sideDiv2) )
  h1third = height / 3.0
  h2third = h1third * 2.0
  f (a,b) = (round a,round b)
  in (map f [(xf, yf - h2third),
              (xf - sideDiv2,yf + h1third),
              (xf + sideDiv2,yf + h1third)],
       map f [(xf - sideDiv2,yf - h1third),
              (xf + sideDiv2,yf - h1third),
              (xf,yf + h2third)])
Now repeat twice and draw

drawStar color1 color2 w side (x,y) = 
  do {  let (a,b) = eqTri side (x,y) 
    ; drawPoly w color1 a 
    ; drawPoly w color2 b 
  }

ex4 =
  runGraphics( 
    do { w <- openWindow "Star of david" 
         (400,400) 
      ; drawStar Red Green w 243 (200,200) 
      ; spaceClose w 
    } )
For a snowflake repeat many times

snowl w color size (x,y) = 
    if size <= minSize
        then return ()
    else do { drawStar color color
                w (fromIntegral size) (x,y)
            ; sequence_ (map smaller allpoints) }
where (triangle1,triangle2) =
    eqTri (fromIntegral size) (x,y)
allpoints = (triangle1 ++ triangle2)
smaller x = snowl w color (size `div` 3) x
To Draw pick appropriate sizes

ex5 =

runGraphics(
    do { w <- openWindow "SnowFlake 1" (400,400)
         ; snow1 w Red 243 (200,200)
         ; spaceClose w
    } )

Why 243?
Multiple Colors

snow2 w colors size (x,y) =
    if size <= minSize
        then return ()
        else do { drawPoly w (colors !! 0) triangle2
        ; drawPoly w (colors !! 1) triangle1
        ; sequence_ (map smaller allpoints)
        }
    where (triangle1,triangle2) = eqTri (fromIntegral size) (x,y)
    allpoints = (triangle1 ++ triangle2)
    smaller x = snow2 w (tail colors) (size `div` 3) x

ex6 =
    runGraphics(
        do { w <- openWindow "Snowflake" (400,400)
            ; snow2 w  (cycle[Red,Blue,Green,Yellow]) 243 (200,200)
            ; spaceClose w
        })
What Happened?

• The list of colors was too short for the depth of the recursion

```haskell
ex6 =
runGraphics(
  do { w <- openWindow "Snowflake 2" (400,400)
       ; snow2 w [Red,Blue,Green,Yellow,White] 243 (200,200)
       ; spaceClose w
   } )

ex7 = runGraphics(
  do { w <- openWindow "Snowflake" (400,400)
       ; snow2 w (cycle [Red,Blue,Green,Yellow])
       243 (200,200)
       ; spaceClose w
   } )
```
snow3 w colors size (x,y) = 
    if size <= minSize
        then return ()
        else do { drawPoly w (colors !! 0) triangle2
                      ; drawPoly w (colors !! 1) triangle1
                      ; snow3 w colors (size `div` 3) (x,y)
                      ; sequence_ (map smaller allpoints) }
    where (triangle1,triangle2) = eqTri (fromIntegral size) (x,y)
         allpoints = (triangle1 ++ triangle2)
         smaller x = snow3 w (tail colors) (size `div` 3) x

ex8 = 
    runGraphics(
        do { w <- openWindow "Snowflake" (400,400)
             ; snow3 w (cycle [Red,Blue,Green,Yellow,White]) 243 (200,200)
             ; spaceClose w } )
Recall the Shape Datatype

data Shape = Rectangle Side Side
  | Ellipse Radius Radius
  | RtTriangle Side Side
  | Polygon [ Vertex ]
deriving Show

type Vertex = (Float,Float)
  -- We call this Vertex (instead of Point) so
  -- as not to confuse it with Graphics.Point

type Side = Float
type Radius = Float
Properties of Shape

- Note that some shapes are position independent
  - Rectangle Side Side
  - RtTriangle Side Side
  - Ellipse Radius Radius

- But the Polygon [Vertex] Shape is defined in terms of where it appears in the plane.

- Shape’s Size and Radius components are measured in inches.

- The Window based drawing mechanism of the last lecture was based upon pixels.
Considerations

- Where to draw position independent shapes?
  - Randomly?
  - In the upper left corner (the window origin)
  - In the middle of the window
- We choose to draw them in the middle of the window.
- We consider this the shape module origin

- So our model has both a different notion of “origin” and of coordinate system (pixels vs inches)

- We need to handle this.
  - Many systems draw about 100 pixels per inch.
Visualize

Window Coordinate system

Shape Coordinate system

(200, 200) pixels or (1, -1) inches
Coercion Functions

\[
\begin{align*}
\text{inchToPixel} & : \text{ Float } \rightarrow \text{ Int } \\
\text{inchToPixel} \ x & = \text{ round } (100 \times x) \\

\text{pixelToInch} & : \text{ Int } \rightarrow \text{ Float } \\
\text{pixelToInch} \ n & = \frac{\text{intToFloat} \ n}{100} \\

\text{intToFloat} & : \text{ Int } \rightarrow \text{ Float } \\
\text{intToFloat} \ n & = \text{ fromInteger } (\text{toInteger} \ n)
\end{align*}
\]
Setting up the Shape window

\[
x_{\text{Win}}, \ y_{\text{Win}} :: \text{Int}
\]
\[
x_{\text{Win}} = 600
\]
\[
y_{\text{Win}} = 500
\]

\[
\text{trans} :: \text{Vertex} \to \text{Point}
\]
\[
\text{trans} (x,y) = ( x_{\text{Win2}} + \text{inchToPixel} \ x, \\
\quad y_{\text{Win2}} - \text{inchToPixel} \ y )
\]

\[
x_{\text{Win2}}, \ y_{\text{Win2}} :: \text{Int}
\]
\[
x_{\text{Win2}} = x_{\text{Win}} \ `\text{div`} \ 2
\]
\[
y_{\text{Win2}} = y_{\text{Win}} \ `\text{div`} \ 2
\]
Translating Points

\[
\text{trans} :: \text{Vertex} \rightarrow \text{Point} \\
\text{trans} \ (x,y) = ( \ x\text{Win2} + \text{inchToPixel} \ x, \\
\quad y\text{Win2} - \text{inchToPixel} \ y )
\]

\[
\text{transList} :: [\text{Vertex}] \rightarrow [\text{Point}] \\
\text{transList} \ [] = [] \\
\text{transList} \ (p:ps) = \text{trans} \ p : \text{transList} \ ps
\]
Translating Shapes

Convert a Shape (Rectangle, Ellipse, ...) into a graphic Draw object (using the window functions line, polygon, ... see file Draw.hs)

\[
\text{shapeToGraphic :: Shape -> Graphic}
\]
\[
\text{shapeToGraphic (Rectangle s1 s2)}
\]
\[
= \text{let } s12 = s1/2
\]
\[
\text{let } s22 = s2/2
\]
\[
\text{in } \text{polygon}
\]
\[
(\text{transList } [(-s12,-s22),(-s12,s22),
\]
\[
(s12,s22),(s12,-s22)])
\]
\[
\text{shapeToGraphic (Ellipse r1 r2)}
\]
\[
= \text{ellipse } (\text{trans } (-r1,-r2)) (\text{trans } (r1,r2))
\]
\[
\text{shapeToGraphic (RtTriangle s1 s2)}
\]
\[
= \text{polygon } (\text{transList } [(0,0),(s1,0),(0,s2)])
\]
\[
\text{shapeToGraphic (Polygon pts)}
\]
\[
= \text{polygon } (\text{transList } pts)
\]

Note, first three are position independent, centered about the origin.
Define some test Shapes

\[
\begin{align*}
\text{sh1, sh2, sh3, sh4} & \ :: \ \text{Shape} \\
\text{sh1} & = \ \text{Rectangle 3 2} \\
\text{sh2} & = \ \text{Ellipse 1 1.5} \\
\text{sh3} & = \ \text{RtTriangle 3 2} \\
\text{sh4} & = \ \text{Polygon [(-2.5,2.5),} \\
& \quad \quad \quad \quad \quad \quad (-1.5,2.0), \\
& \quad \quad \quad \quad \quad \quad (-1.1,0.2), \\
& \quad \quad \quad \quad \quad \quad (-1.7,-1.0), \\
& \quad \quad \quad \quad \quad \quad (-3.0,0)]
\end{align*}
\]
Draw a Shape

ex9

= runGraphics (do w <- openWindow "Drawing Shapes" (xWin,yWin)
drawInWindow w
    (withColor Red (shapeToGraphic sh1))
drawInWindow w
    (withColor Blue (shapeToGraphic sh2))
spaceClose w)
Draw multiple Shapes

type ColoredShapes = [(Color,Shape)]

shs :: ColoredShapes
shs = [(Red,sh1),(Blue,sh2),
       (Yellow,sh3),(Magenta,sh4)]

drawShapes :: Window -> ColoredShapes -> IO ()
drawShapes w [] = return ()
drawShapes w ((c,s):cs)
    = do drawInWindow w
         (withColor c (shapeToGraphic s))
         drawShapes w cs
ex10

= runGraphics (
do w <- openWindow
    "Drawing Shapes" (xWin,yWin)
drawShapes w shs
spaceClose w )
Another Example

ex11
    = runGraphics (  
        do w <- openWindow "Drawing Shapes" (xWin,yWin)  
            drawShapes w (reverse coloredCircles)  
            spaceClose w  
    )

conCircles = map circle [0.2,0.4 .. 1.6]

coloredCircles =  
    zip [Black, Blue, Green, Cyan, Red, Magenta, Yellow, White]  
        conCircles
The Region datatype

• A region represents an area on the two dimensional plane
• Its represented by a tree-like data-structure

```
-- A Region is either:
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
```
Regions and Trees

• Why is Region tree-like?

• What’s 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 make infinite regions?

• What does a region mean?
The Region datatype

• A region represents an area on the two dimensional plane

-- A Region is either:

    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)
Why Regions?

• Regions are interesting because
  – They allow us to build complicated “shapes” from simple ones
  – They illustrate the use of tree-like data structures
    • What makes regions tree-like?
  – They “solve” the problem of only having rectangles and ellipses centered about the origin.
  – They make a beautiful analogy with mathematical sets
What is a region?

• A Region is all those points that lie within some area in the 2 dimensional plane.
• This often (almost always?) an infinite set.
• An efficient representation is as a characteristic function.

• What do they look like? What do they represent?
Translate \((u,v)\) \(r\)
scale \((x, y)\) \(r\) =
\[
\begin{bmatrix}
(a \times x, b \times y) & | & (a, b) \leftarrow r
\end{bmatrix}
\]
Complement r
Region Characteristic functions

• We define the meaning of a region by its characteristic function.

\[
\text{containsR} :: \text{Region} \to \text{Coordinate} \to \text{Bool}
\]

• How would you write this function?
  – Recursion, using pattern matching over the structure of a Region
  – What are the base cases of the recursion?

• Start with a characteristic function for a primitive Shape
(Rectangle s1 s2) `containsS` (x,y)
  = let t1 = s1/2
     t2 = s2/2
     in -t1<=x && x<=t1 && -t2<=y && y<=t2
(Ellipse $r_1$ $r_2$) \ `containsS` \ (x,y) \\
= \((x/r_1)^2 + (y/r_2)^2 \leq 1\)
Left of a line that bisects the plane

For a Ray specified by two points \((A, B)\), and facing in the direction from \(A\) to \(B\), a Vertex \((px,py)\) is to the left of the line when:

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

\[
(px,py) \ `\text{isLeftOf}` `\text{((ax,ay),(bx,by))} = \text{let } (s,t) = (px-ax, py-ay)
\]

\[
(u,v) = (px-bx, py-by)
\]

\[
in \ s*v >= t*u
\]
Inside a (Convex) Polygon

A Vertex, P, is inside 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 isLeftOffp(zip pts shiftpts)
       isLeftOffp p' = isLeftOfp p p'
       in foldr (&&) True leftOfList
RtTriangle

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

\[
\begin{array}{c}
  (0,s2) \\
  \downarrow \\
  \downarrow \\
  \downarrow \\
  (s1,0) \\
\end{array}
\]
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 foldr (&&) True leftOfList
(RtTriangle s1 s2) `containsS` p
    = (Polygon [(0,0),(s1,0),(0,s2)]) `containsS` p
containsR using patterns

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
Pictures

• Drawing Pictures
  – Pictures are composed of Regions
    • Regions are composed of shapes
  – Pictures add Color

data Picture = Region Color Region
  | Picture `Over` Picture
  | EmptyPic
  deriving Show

Must be careful to use SOEGraphics, but
SOEGraphics has its own Region datatype.

import SOEGraphics hiding (Region)
import qualified SOEGraphics as G (Region)
Recall our Region datatype

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

How will we draw things like the intersection of two regions, or the complement of two regions. These are hard things to do, and require hardware support to do efficiently. The G.Region type interfaces to this hardware support.
G.Region

• The G.Region datatype interfaces to the hardware. It is essentially a two dimensional array or “bit-map”, storing a binary value for each pixel in the window.
Hardware support

- There is efficient hardware support for combining two bit-maps using binary operators.

- Operations are fast, but data (space) intensive, and this space needs to be explicitly allocated and de-allocated.
Interface

createRectangle :: Point -> Point -> IO G.Region
createEllipse   :: Point -> Point -> IO G.Region
createPolygon   :: [Point] -> IO G.Region

andRegion       :: G.Region -> G.Region -> IO G.Region
orRegion        :: G.Region -> G.Region -> IO G.Region
xorRegion       :: G.Region -> G.Region -> IO G.Region
diffRegion      :: G.Region -> G.Region -> IO G.Region
deleteRegion    :: G.Region -> IO ()

drawRegion      :: G.Region -> Graphic

These functions are defined in the SOE library module.
Drawing G.Region

• To draw things quickly, turn them into a G.Region, then turn the G.Region into a graphic object and then use all the machinery we have built up so far.

```haskell
drawRegionInWindow :: Window -> Color -> Region -> IO ()

drawRegionInWindow w c r =
    drawInWindow w
    (withColor c (drawRegion (regionToGRegion r)))
```

• All we need to define then is:
  `regionToGRegion`
  – we’ll come back to `regionToGRegion` in a minute
Drawing Pictures

• Pictures combine multiple regions into one big picture. They provide a mechanism for placing one sub-picture on top of another.

drawPic :: Window -> Picture -> IO ()

drawPic w (Region c r)   = drawRegionInWindow w c r
drawPic w (p1 `Over` p2) = do { drawPic w p2
                                   ; drawPic w p1
                                 }
drawPic w EmptyPic       = return ()
Overview

- We have a rich calculus of *Shapes*, which we can draw, take the perimeter of, and tell if a point lies within.
- We extend this with a richer type *Region*, which allows more complicated ways of combination (intersection, complement, etc.).
  - We gave *Region* a mathematical semantics as a set of points in the 2-dimensional plane.
  - We defined some interesting operators like `containsR` which is the characteristic function for a region.
  - The rich combination ability make *Region* hard to draw efficiently, so we use a lower level datatype supported by the hardware: `G.Region` which is essentially a bit-map.
- We enrich this even further with the *Picture* type.
- `G.Region` is low level, relying on features like overwriting, and explicit allocation and deallocation of memory.
  - We think of *Region*, as a highlevel interface to `G.Region` which hides the low level details.
Turning a Region into a G.Region

Experiment with a smaller problem to illustrate a lurking efficiency problem.

data NewRegion = Rect Side Side  -- Abstracts G.Region

regToNReg1 :: Region -> NewRegion
regToNReg1 (Shape (Rectangle sx sy))
    = Rect sx sy
regToNReg1 (Scale (x,y) r)
    = regToNReg1 (scaleReg (x,y) r)
where scaleReg (x,y) (Shape (Rectangle sx sy))
    = Shape (Rectangle (x*sx) (y*sy))
scaleReg (x,y) (Scale s r)
    = Scale s (scaleReg (x,y) r)

Note, scaleReg distributes over Scale
Problem

• Consider

(Scale (x1,y1)
  (Scale (x2,y2)
    (Scale (x3,y3)
      ...
      (Shape (Rectangle sx sy))
    ...
))

• If the Scale level is N-deep, how many traversals does \text{regToNReg1} do of the Region tree?
You’ve probably seen this before

• Believe it or not you probably have encountered this problem before. Recall the definition of reverse

\[
\text{reverse} \; [\;] = [\;] \\
\text{reverse} \; (x:xs) = (\text{reverse} \; xs) \; ++ \; [x] \\
\quad \text{where} \; [\;] \; ++ \; zs = zs \\
\quad \quad (y:ys) \; ++ \; zs = y : (ys \; ++ \; zs)
\]

• How did we solve this? Use an extra accumulating parameter.

\[
\text{reverse} \; xs = \text{revhelp} \; xs \; [\;] \\
\quad \text{where} \; \text{revhelp} \; [\;] \; zs = zs \\
\quad \quad \text{revhelp} \; (x:xs) \; zs = \text{revhelp} \; xs \; (x:zs)
\]
Accumulate a complex Scale

\[ \text{regToNReg2} :: \text{Region} \to \text{NewRegion} \]
\[ \text{regToNReg2} \; r = \text{rToNR} \; (1,1) \; r \]
where \[ \text{rToNR} :: (\text{Float},\text{Float}) \to \text{Region} \to \text{NewRegion} \]
\[ \text{rToNR} \; (x_1,y_1) \; (\text{Shape} \; (\text{Rectangle} \; x_1 \; y_1)) \]
\[ = \text{Rect} \; (x_1 \times x_2) \; (y_1 \times y_2) \]
\[ \text{rToNR} \; (x_1,y_1) \; (\text{Scale} \; (x_2,y_2) \; r) \]
\[ = \text{rToNR} \; (x_1 \times x_2,y_1 \times y_2) \; r \]

• To solve our original problem Repeat this for all the constructors of \text{Region} (not just \text{Shape} and \text{Scale}) and use \text{G.Region} instead of \text{NewRegion}, We also need to handle translation as well as scaling
Final Version

```haskell
regToGReg1 :: Vector -> Vector -> Region -> G.Region
regToGReg1 trans sca (Shape s) = shapeToGRegion trans sca s
regToGReg1 (x,y) sca (Translate (u,v) r)
    = regToGReg1 (x+u, y+v) sca r
regToGReg1 trans (x,y) (Scale (u,v) r)
    = regToGReg1 trans (x*u, y*v) r
regToGReg1 trans sca Empty = createRectangle (0,0) (0,0)
regToGReg1 trans sca (r1 `Union` r2)
    = let gr1 = regToGReg1 trans sca r1
        gr2 = regToGReg1 trans sca r2
        in orRegion gr1 gr2

• Assuming of course we can write:

    shapeToGRegion :: Vector -> Vector -> Shape -> G.Region

and write rules for Intersect, Complement etc.
A matter of style

• While the function on the previous page shows how to solve the problem, there are several stylistic issues that could make it more readable and understandable.

• The style of defining a function by patterns, becomes cluttered when there are many parameters (other than the one which has the patterns).

• The pattern of explicitly allocating and deallocating (bit-map) G.Region’s will be repeated in cases for intersection and for complement, so we should abstract it, and give it a name.
Abstract the low level bit-map details

primGReg trans sca r1 r2 op
    = let gr1 = regToGReg trans sca r1
       gr2 = regToGReg trans sca r2
    in op gr1 gr2
Redo with a case expression

\[
\text{regToGReg :: Vector -> Vector -> Region -> G.Region}
\]

\[
\text{regToGReg (trans @ (x,y)) (sca @ (a,b)) shape =}
\]

case shape of

(Shape s) -> \text{shapeToGRegion trans sca s}

(Translate (u,v) r) -> \text{regToGReg (x+u, y+v) sca r}

(Scale (u,v) r) -> \text{regToGReg trans (a*u, b*v) r}

(Empty) -> \text{createRectangle (0,0) (0,0)}

(r1 `Union` r2) -> \text{primGReg trans sca r1 r2 orRegion}

(r1 `Intersect` r2) -> \text{primGReg trans sca r1 r2 andRegion}

(Complement r) -> \text{primGReg trans sca winRect r diffRegion}

where \text{winRect :: Region}

\[
\text{winRect = Shape (Rectangle (pixelToInch xWin) (pixelToInch yWin))}
\]

\[
\text{regionToGRegion :: Region -> G.Region}
\]

\[
\text{regionToGRegion r = regToGReg (0,0) (1,1) r}
\]

Pattern renaming
Shape to G.Region: Rectangle

\[ xWin2 = \frac{xWin}{2} \]
\[ yWin2 = \frac{yWin}{2} \]

\[
\text{shapeToGRegion1} :: \text{Vector} \rightarrow \text{Vector} \rightarrow \text{Shape} \rightarrow \text{IO G.Region}
\]
\[
\text{shapeToGRegion1} \ (lx,ly) \ (sx,sy) \ (\text{Rectangle} \ s1 \ s2)
\]
\[
= \text{createRectangle} \ \left( \text{trans}(-s1/2,-s2/2) \ (\text{trans} \ (s1/2,s2/2)) \right)
\]
\[
\text{where} \ \text{trans} \ (x,y) = (xWin2 + \text{inchToPixel} \ ((x+lx)*sx), \ yWin2 - \text{inchToPixel} \ ((y+ly)*sy))
\]
Ellipse

\[
\text{shapeToGRegion1} \ (lx, ly) \ (sx, sy) \ (\text{Ellipse} \ r1 \ r2) \\
= \text{createEllipse} \ (\text{trans} \ (-r1, -r2)) \ (\text{trans} \ (r1, r2)) \\
\text{where} \ \text{trans} \ (x, y) = \\
( \ xWin2 + \ \text{inchToPixel} \ ((x+lx)*sx), \\
\ yWin2 - \ \text{inchToPixel} \ ((y+ly)*sy) )
\]
Polygon and RtTriangle

\[
\text{shapeToGRegion1} \ (lx, ly) \ (sx, sy) \ \text{(Polygon pts)} \\
= \text{createPolygon} \ (\text{map} \ \text{trans} \ \text{pts}) \\
\text{where} \ \text{trans} \ (x, y) = \\
\quad (\ xWin2 + \text{inchToPixel} \ ((x+lx)*sx), \ \\
\quad yWin2 - \text{inchToPixel} \ ((y+ly)*sy) \ ) \\
\text{shapeToGRegion1} \ (lx, ly) \ (sx, sy) \ \text{(RtTriangle s1 s2)} \\
= \text{createPolygon} \ (\text{map} \ \text{trans} \ [(0,0),(s1,0),(0,s2)]) \\
\text{where} \ \text{trans} \ (x, y) = \\
\quad (\ xWin2 + \text{inchToPixel} \ ((x+lx)*sx), \ \\
\quad yWin2 - \text{inchToPixel} \ ((y+ly)*sy) \ )
\]
A matter of style, again

• shapeToGRegion1 has the same problems as regToGReg1
  – The extra translation and scaling parameters obscure the pattern matching
  – There is a repeated pattern, we should give it a name.

```haskell
shapeToGRegion (lx,ly) (sx,sy) s =
  case s of
    Rectangle s1 s2 -> createRectangle
      (trans (-s1/2,-s2/2))
      (trans (s1/2,s2/2))
    Ellipse r1 r2 -> createEllipse
      (trans (-r1,-r2))
      (trans ( r1, r2))
    Polygon pts -> createPolygon (map trans pts)
    RtTriangle s1 s2 -> createPolygon
      (map trans [(0,0),(s1,0),(0,s2)])
  where trans (x,y) = ( xWin2 + inchToPixel ((x+lx)*sx),
                      yWin2 - inchToPixel ((y+ly)*sy) )
```
## Drawing Pictures, Sample Regions

```haskell
draw :: Picture -> IO ()
draw p
    = runGraphics (     
    do w <- openWindow "Region Test" (xWin,yWin)
        drawPic w p
        spaceClose w
    )

r1 = Shape (Rectangle 3 2)
r2 = Shape (Ellipse 1 1.5)
r3 = Shape (RtTriangle 3 2)
r4 = Shape (Polygon [(-2.5,2.5), (-3.0,0),
                      (-1.7,-1.0),
                      (-1.1,0.2), (-1.5,2.0)])
```
Sample Pictures

\[ \text{reg1} = \text{r3} \quad \text{`Union`} \quad \text{--RtTriangle} \]
\[ \text{r1} \quad \text{`Intersect`} \quad \text{-- Rectangle} \]
\[ \text{Complement r2} \quad \text{`Union`} \quad \text{-- Ellipse} \]
\[ \text{r4} \quad \text{-- Polygon} \]

\[ \text{pic1} = \text{Region Cyan reg1} \]

\[ \text{ex12} = \text{draw} \]
\[ \quad \text{“first region picture”} \]
\[ \quad \text{pic1} \]

Recall the precedence of Union and Intersect
More Pictures

reg2 = let circle = Shape (Ellipse 0.5 0.5)
    square = Shape (Rectangle 1 1)
    in (Scale (2,2) circle)
    `Union` (Translate (2,1) square)
    `Union` (Translate (-2,0) square)

pic2 = Region Yellow reg2

ex13 =
    draw "Ex 13" pic2
Another Picture

pic3 = pic2 `Over` pic1

ex14 = draw “ex14” pic3
Separate computation from action

oneCircle = Shape (Ellipse 1 1)
manyCircles = [ Translate (x,0) oneCircle | x <- [0,2..] ]
fiveCircles = foldr Union Empty (take 5 manyCircles)
pic4 = Region Magenta (Scale (0.25,0.25) fiveCircles)
ex15 = draw “Ex15” pic4
Ordering Pictures

pictToList :: Picture -> [(Color,Picture.Region)]

pictToList EmptyPic = []
pictToList (Region c r) = [(c,r)]
pictToList (p1 `Over` p2)
    = pictToList p1 ++ pictToList p2

pic6 = pic4 `Over` pic2 `Over` pic1 `Over` pic3

Recovers the Regions from top to bottom possible because Picture is a datatype that can be analysed
An Analogy

pictToList EmptyPic = []
pictToList (Region c r) = [(c,r)]
pictToList (p1 `Over` p2) = pictToList p1 ++ pictToList p2

drawPic w (Region c r) = drawRegionInWindow w c r
drawPic w (p1 `Over` p2) = do { drawPic w p2 ; drawPic w p1}
drawPic w EmptyPic = return ()

• Something to prove:
  sequence .
  (map (uncurry (drawRegionInWindow w))) . Reverse . pictToList = drawPic w
Pictures that React

- Find the Topmost Region in a picture that “covers” the position of the mouse when a left button click appears.
- Search the picturelist for the first Region that contains the mouse position.
- Re-arrange the list, bring that one to the top

```haskell
adjust :: [(Color, Picture.Region)] -> Vertex ->
        (Maybe (Color, Picture.Region), [(Color, Picture.Region)])

adjust []       p = (Nothing, [])
adjust ((c,r):regs) p =
    if r `containsR` p
    then (Just (c,r), regs)
    else let (hit, rs) = adjust regs p
          in  (hit, (c,r) : rs)
```
Doing it Non-recursively

adjust2 regs p = case (break \(_,r) -> r `containsR` p) regs of
 (top,hit:rest) -> (Just hit, top++rest)
 (_,[])         -> (Nothing, [])

break:: (a -> Bool) -> [a] -> ([a],[a])
is from the Prelude.

Break even [1,3,5,4,7,6,12]
([1,3,5],[4,7,6,12])
Putting it all together

```
loop :: Window -> [(Color, Picture.Region)] -> IO ()
loop w regs =
do clearWindow w
    sequence [ drawRegionInWindow w c r |
              (c,r) <- reverse regs ]
    (x,y) <- getLBP w
    case (adjust regs (pixelToInch (x - (xWin `div` 2)),
                 pixelToInch ((yWin `div` 2) - y)) of
         (Nothing, _      ) -> closeWindow w
         (Just hit, newRegs) -> loop w (hit : newRegs)

draw2 :: Picture -> IO ()
draw2 pic
    = runGraphics (do w <- openWindow "Picture demo" (xWin,yWin)
                    loop w (pictToList pic))
```
Try it out

p1, p2, p3, p4 :: Picture
p1 = Region Magenta r1
p2 = Region Cyan r2
p3 = Region Green r3
p4 = Region Yellow r4

pic :: Picture
pic = foldl1 Over EmptyPic [p1, p2, p3, p4]
main = draw2 pic
A matter of style, 3

```
loop2 w regs
    = do clearWindow w
        sequence [ drawRegionInWindow w c r | (c,r) <- reverse regs ]
        (x,y) <- getLBP w
        let aux (_,r) = r `containsR`
            ( pixelToInch (x-xWin2),
              pixelToInch (yWin2-y) )
        case (break aux regs) of
            (_,[])        -> closeWindow w
            (top,hit:bot) -> loop w (hit : (top++bot))

draw3 :: Picture -> IO ()
draw3 pic
    = runGraphics (  
        do w <- openWindow "Picture demo" (xWin,yWin)  
           loop2 w (pictToList pic) )
```