CS 457/557 Functional Programming

Lecture 10x
Drawing Regions
Pictures

• Drawing Pictures
  – Pictures are composed of Regions
    Regions are composed of shapes
  – Pictures add color and layering

\[
data \text{ Picture} = \text{ Region Color Region} \\
\quad \text{Picture `Over` Picture} \\
\quad \text{EmptyPic} \\
\quad \text{deriving Show}
\]

• We need to use SOEGraphics, but SOEGraphics has its own Region datatype.

\[
\text{import SOEGraphics hiding (Region)} \\
\text{import qualified SOEGraphics as G (Region)}
\]
Recall the \textbf{Region} Datatype

\begin{verbatim}
data Region =  
  Shape Shape                -- primitive shape  
  Translate Vector Region    -- translated region 
  Scale Vector Region        -- scaled region    
  Complement Region          -- inverse of a region 
  Region `Union` Region      -- union of regions 
  Region `Intersect` Region  -- intersection of regions 
  Empty                      
\end{verbatim}

• How do we draw things like the intersection of two regions, or the complement of a region? These are hard things to do efficiently. Fortunately, the \texttt{G.Region} interface uses lower-level support to do this for us.
The **G.Region** datatype interfaces more directly to the underlying hardware. It is essentially a two-dimensional array or “bit-map”, storing a binary value for each pixel in the window.
Efficient Bit-Map Operations

- There is efficient low-level support for combining bit-maps using a variety of operators. For example, for union:

- These operations are fast, but data (space) intensive, and this space needs to be explicitly allocated and de-allocated, a job for a much lower-level language.
G.Region Interface

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

andRegion        :: G.Region -> G.Region -> G.Region  
orRegion         :: G.Region -> G.Region -> G.Region  
xorRegion        :: G.Region -> G.Region -> G.Region  
diffRegion       :: G.Region -> G.Region -> G.Region

drawRegion       :: G.Region -> Graphic

These functions are defined in the SOEGraphics 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 to display the object.

```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.
  - See textbook for details.
  - Meanwhile, let’s define what it means to draw a picture.
Drawing Pictures

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

\[
\text{drawPic} :: \text{Window} \rightarrow \text{Picture} \rightarrow \text{IO} ()
\]

\[
\begin{align*}
\text{drawPic} w \ (\text{Region} \ c \ r) &= \text{drawRegionInWindow} w c r \\
\text{drawPic} w \ (\text{p1 `Over` p2}) &= \text{do} \ \text{drawPic} w \ p2 \ \text{drawPic} w \ p1 \\
\text{drawPic} w \ \text{EmptyPic} &= \text{return} ()
\end{align*}
\]

- Note that \( p2 \) is drawn before \( p1 \), since we want \( p1 \) to appear “over” \( p2 \).
Summary

- We have a rich calculus of Shapes, which we can draw, take the perimeter of, and tell if a point lies within.
- We defined a richer data type Region, which allows more complex compositions (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 nature of Region makes it hard to draw efficiently, so we use a lower level datatype G.Region, which relies on features like overwriting and explicit allocation and deallocation of memory.
  - We can think of Region as a high-level interface to G.Region that hides low-level details.
- We enriched things even further with the Picture type, which adds color and layering.
Drawing Pictures, Sample Regions

draw :: Picture \rightarrow 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

\[
\begin{align*}
\text{reg1} &= r3 \ `\text{Union}` \\
\text{r1} &= r3 \ `\text{Union}` \\
\text{Intersect} &= r3 \\
\text{Complement} &= r2 \ `\text{Union}` \\
\text{r4} &= r2 \ `\text{Union}` \\
\end{align*}
\]

---

\[
\begin{align*}
\text{RtTriangle} &= \text{Rectangle} \\
\text{Rectangle} &= \text{Ellipse} \\
\text{Ellipse} &= \text{Polygon} \\
\end{align*}
\]

\[
\begin{align*}
\text{pic1} &= \text{Region Cyan reg1} \\
\text{Main1} &= \text{draw pic1} \\
\end{align*}
\]

Recall the precedence of Union and Intersect
More Pictures

\[
\begin{align*}
\text{reg2} &= \text{let}\ \text{circle} = \text{Shape (Ellipse 0.5 0.5)} \\
&\quad \text{square} = \text{Shape (Rectangle 1 1)} \\
&\quad \text{in} (\text{Scale (2,2) circle}) \\
&\quad \text{`Union` (Translate (2,1) square)} \\
&\quad \text{`Union` (Translate (-2,0) square)} \\
\text{pic2} &= \text{Region Yellow reg2} \\
\text{main2} &= \text{draw pic2}
\end{align*}
\]
Another Picture

pic3 = pic2 `Over` pic1
main3 = draw 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)
main4 = draw pic4
Ordering Pictures

\[
pictToList :: \text{Picture} \rightarrow [\text{(Color, Region)}]
\]

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

\[
pic6 = pic4 \ `Over` pic2 \ `Over` pic1 \ `Over` pic5
\]
\[
pictToList pic6 \longrightarrow
\]
\[
[(\text{Magenta,?}), (\text{Yellow,?}), (\text{Cyan,?}), (\text{Cyan,?})]
\]

Recovers the Regions from top to bottom.
Possible because Picture is a datatype that can be analyzed.
Two ways of drawing a picture

\[\text{pictToList } \text{EmptyPic} \quad = \quad []\]
\[\text{pictToList } \text{(Region } c \quad r) \quad = \quad [(c,r)]\]
\[\text{pictToList } \text{(p1 `Over` p2)} \quad = \quad \text{pictToList p1} \quad +\quad \text{pictToList p2}\]

\[\text{drawPic w } \text{(Region } c \quad r) \quad = \quad \text{drawRegionInWindow w } c \quad r\]
\[\text{drawPic w } \text{(p1 `Over` p2)} \quad = \quad \text{do} \quad \{\quad \text{drawPic w p2} \quad \}
\quad ;\quad \text{drawPic w p1}\]
\[\text{drawPic w } \text{EmptyPic} \quad = \quad \text{return } ()\]

- Something interesting to prove:
  \[\text{drawPic w} \quad = \quad \text{sequence} \quad .\]
  \[\text{(map } (\text{uncurry} \quad (\text{drawRegionInWindow} \quad w))\quad .\]
  \[\text{reverse} \quad .\]
  \[\text{pictToList}\]
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 picture list for the first Region that contains the mouse position.
- Re-arrange the list, bringing that one to the top.

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

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

This is from the Prelude:

```haskell
break:: (a -> Bool) -> [a] -> ([a],[a])
```

For example:

```haskell
break even [1,3,5,4,7,6,12] = ([1,3,5],[4,7,6,12])
```
Putting it all Together

\[ \text{rtrans} :: \text{Point} \rightarrow \text{Vertex} \]
\[ \text{rtrans} (x, y) = (\text{pixelToInch} (x - x\text{Win}2), \]
\[ \quad \text{pixelToInch} (y\text{Win}2 - y)) \]

\[ \text{loop w regs} \]
\[ \quad = \text{do clearWindow w} \]
\[ \quad \quad \text{sequence [ drawRegionInWindow w c r |} \]
\[ \quad \quad \quad (c, r) \leftarrow \text{reverse regs} \] \]
\[ \quad \quad (x, y) \leftarrow \text{getLBP w} \]
\[ \quad \text{let aux (_, r) = r \ `containsR` \ (rtrans (x, y))} \]
\[ \quad \text{case (break aux regs) of} \]
\[ \quad \quad (_, []) \rightarrow \text{closeWindow w} \]
\[ \quad \quad (\text{top, hit: bot}) \rightarrow \text{loop w (hit : (top++bot))} \]

\[ \text{draw pic} = \text{runGraphics (} \]
\[ \quad \text{do w \leftarrow openWindow "Picture demo" (x\text{Win}, y\text{Win})} \]
\[ \quad \quad \text{loop w (pictToList pic)} \]
\[ \quad \text{)} \]