Functional Graphics in Scala

Bernie Pope
Melbourne Scala Users Group
May 26 2014

Inspiration

This talk is inspired by:

Functional Images by Conal Elliot
http://conal.net/papers/functional-images/fop-conal.pdf

which appeared in, The Fun of Programming (2003).

The original was in Haskell.

Standard image representation

Computer graphic images are usually represented as two-dimensional arrays of pixels:

```
   import java.awt.{Color}

   type Image = Array[Array[Color]]
```

Functional graphics

A more abstract representation:

```
   type Image[T] = (Double, Double) => T
```

- Defined (infinitely) over the two-dimensional real coordinate space.
- Parameterised over “pixel” type `T`.

Simple example: constant image

```
def constImage[T](value:T):Image[T] = (_, _) => value

val redImage = constImage(Color.red)
```

Viewing an image

```
class Display(cols:Int, rows:Int) {
   private val buffer = …

   ...

   def rasterize(image:Image[Color]) = {
      for (x <- 0 to cols - 1;
           y <- 0 to rows - 1)
         buffer.setRGB(x, y, image(x, y).getRGB)
   }

   def open() = ...
}
```

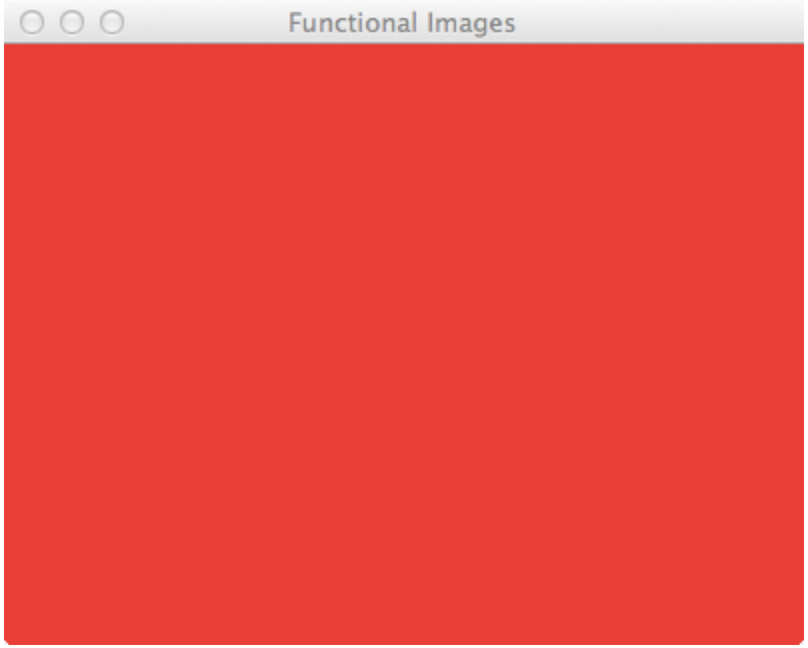
Viewing an image

```
class Draw(cols:Int, rows:Int, image:Image[Color]) {

   def show() = {
      val display = new Display(cols, rows)
      display.rasterize(image)
      display.open()
   }
}
```

Viewing the red image

```
new Draw(400, 300, redImage).show()
```



Fancy example: black and white grid

```
// Modulus which returns a positive result, even for
// negative numerators.
def modDouble(x:Double, y:Double):Double = {
      val m = x % y
      if (m < 0) m + y else m
}

def grid(cell:Double, line:Double):Image[Boolean] = {
   (col, row) => (modDouble(col, cell) >= line &&
                  modDouble(row, cell) >= line)
}
```

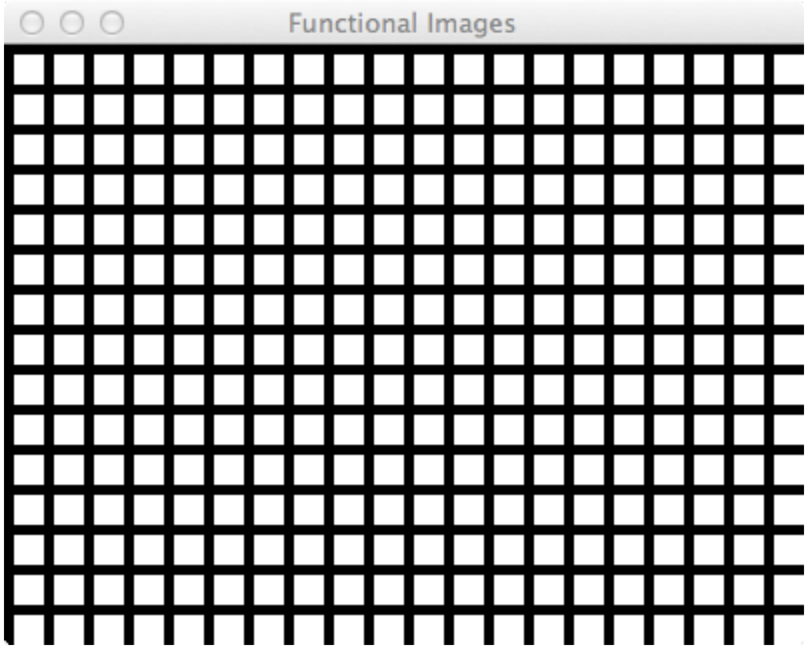
Problem: how to display Boolean image?

```
def mapImage[A,B](fun:A => B, image:Image[A]):Image[B] =
      (col, row) => fun(image(col, row))

implicit
   def BoolToColor(image:Image[Boolean]):Image[Color] =
      mapImage((value:Boolean) =>
         if (value) Color.white else Color.black, image)
```

Viewing a grid

```
val gridImage = grid(20, 5)
new Draw(400, 300, gridImage).show()
```



An observation about complexity

- We only walk over the pixels of the output image once, within `rasterize()`.

Image transformations

```
type ImageTrans[T] = Image[T] => Image[T]
type CoordTrans = (Double, Double) => (Double, Double)

def coordTrans[T](trans:CoordTrans):ImageTrans[T] =
   (image:Image[T]) =>
      (col:Double, row:Double) =>
         image.tupled(trans(col, row))

def translate[T](colD:Double, rowD:Double):ImageTrans[T] =
   coordTrans((col, row) => (col - colD, row - rowD))
```

Image transformations

```
def rotateOrigin[T](angle:Double):ImageTrans[T] = {
   val cosAngle = cos(angle)
   val sinAngle = sin(angle)
   coordTrans(
         (col, row) =>
            (col * cosAngle - row * sinAngle,
             col * sinAngle + row * cosAngle))
}
```

Image transformations

```
def aboutPoint[T](transform:ImageTrans[T],
                  col:Double, row:Double):ImageTrans[T] =
   translate(-col, -row) andThen
   transform andThen
   translate(col, row)

def rotate[T](angle:Double,
              col:Double, row:Double):ImageTrans[T] =
   aboutPoint(rotateOrigin(angle), col, row)
```

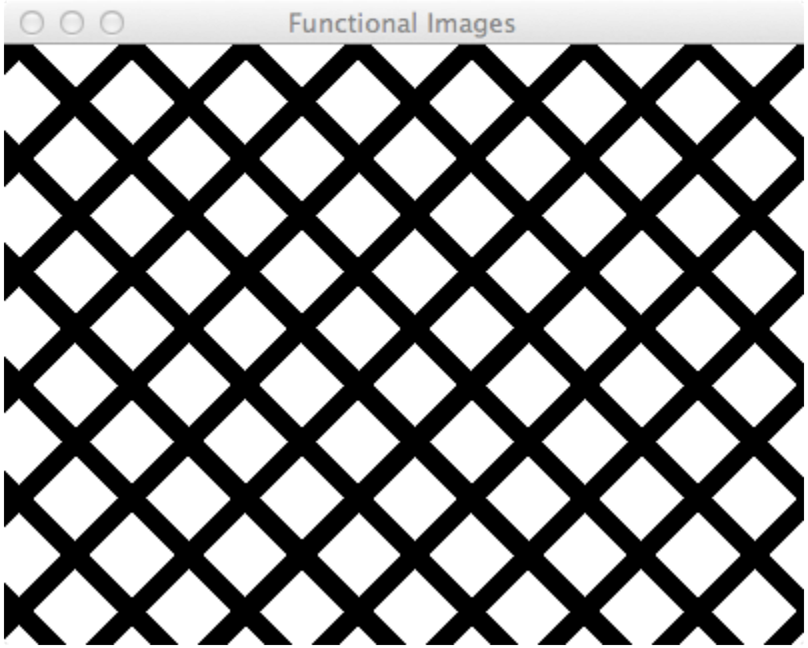
Image transformations

```
def scaleOrigin[T](factor:Double):ImageTrans[T] =
   coordTrans((col, row) => (col / factor, row / factor))


def scale[T](factor:Double,
             col:Double, row:Double):ImageTrans[T] =
   aboutPoint(scaleOrigin(factor), col, row)
```

Applying transformations

```
def scaleRotate[T](s:Double, a:Double):ImageTrans[T] =
   scaleOrigin(s) andThen rotateOrigin(a)
val scaledRotatedGrid = scaleRotate(2, Pi/4)(gridImage)
new Draw(400, 300, scaledRotatedGrid).show()
```

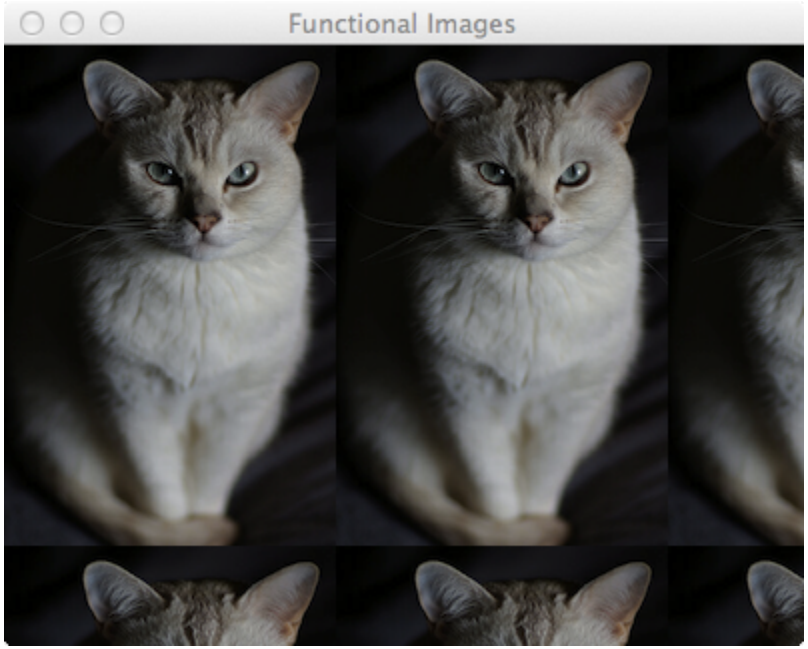


Loading a bitmap from file

```
def bitmap(filepath:String):Image[Color] = {
   val pixels = ImageIO.read(new File(filepath))
   val numRows = pixels.getHeight
   val numCols = pixels.getWidth
   (col:Double, row:Double) => {
       val colInt = modInt(col.toInt, numCols)
       val rowInt = modInt(row.toInt, numRows)
       new Color(pixels.getRGB(colInt, rowInt))
   }
}
```

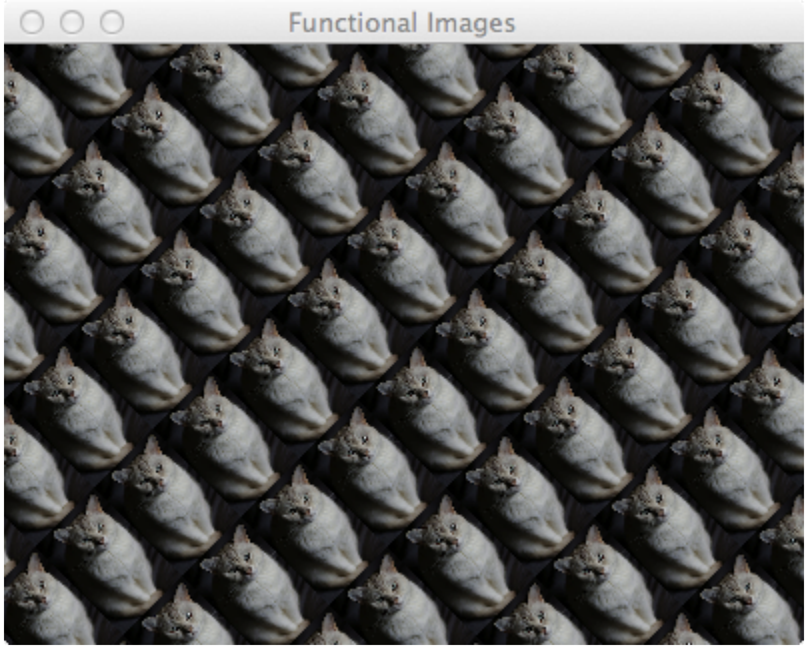
Loading a bitmap from file

```
val bitmapImage = bitmap("floyd.png")
new Draw(400, 300, bitmapImage).show()
```



Transforming a bitmap

```
val scaledRotatedBitmap =
   scaleRotate(0.25, Pi/4)(bitmapImage)
new Draw(400, 300, bitmapImage).show()
```



Waves

```
def waveIntensityOrigin(phase: Double, vert:Double,
         amp:Double, period:Double):Image[Double] = {
   val compress = 2 * Pi / period
   val phaseFactor = phase * compress
   (col:Double, row:Double) => {
      val d = distance(col, row, 0, 0)
      amp * + cos(compress * d - phaseFactor) + vert
   }
}
def waveIntensity(phase: Double, vert:Double, amp:Double,
      period:Double, col:Double, row:Int):Image[Double] =
   translate(col, row)(waveIntensityOrigin(phaseShift,
       vertShift, amp, period))
```

Waves

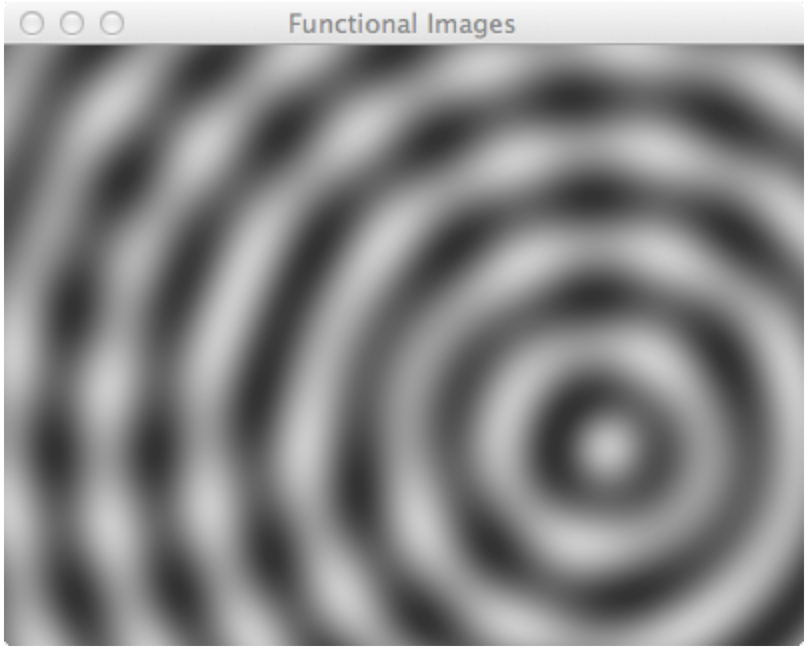
```
implicit
   def DoubleToColor(image:Image[Double]):Image[Color] =
      mapImage((value:Double) => {
         val intensity = clampIntensity((value * 255).toInt)
         new Color(intensity, intensity, intensity)}, image)

def combineImage[A,B,C](image1:Image[A], image2:Image[B],
                        combine:(A, B) => C):Image[C] =
   (col, row) => combine(image1(col, row), image2(col, row))

def waveImage:Image[Double] = {
   val wave1 = waveIntensity(0, 0.3, 0.2, 50, 300, 200)
   val wave2 = waveIntensity(0, 0.2, 0.1, 70, 50, 100)
   combineImage(wave1, wave2, (x:Double, y:Double) => x + y)
}
```

Waves

```
new Draw(400, 300, waveImage).show()
```



Animation

We can represent animations as functions from “time” to images:

```
type Animation[T] = Double => Image[T]
```

We allow ourselves a very liberal interpretation of time.

Waves over time

A slight generalisation of the wave image:

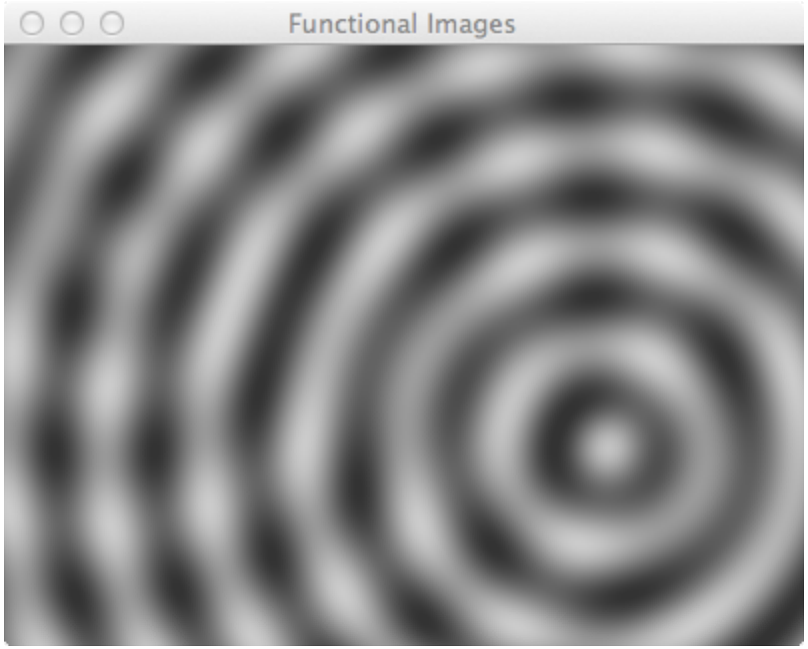
```
def waveAnimation(time:Double):Image[Double] = {
  val wave1 = waveIntensity(time*6, 0.3, 0.2, 50, 300, 200)
  val wave2 = waveIntensity(time*2, 0.2, 0.1, 70, 50, 100)
  combineImage(wave1, wave2, (x:Double, y:Double) => x + y)
}
```

Rendering animations

```
class Animate(cols:Int, rows:Int, animation:Animation[Color])
{
   def show() = {
      val display = new Display(cols, rows)
      var time = 0.0
      var timeDelta = 1.0
      display.open()
      while(true) {
         display.rasterize(animation(time))
         display.repaint()
         time += timeDelta
      }
   }
}
```

Showing the waves

```
new Animate(400, 300, waveAnimation).show()
```



above is a still shot from the animation

Fancy animation

Modulate the scale of an image based on the wave function.

```
def waveScaleOrigin[T](phase: Double, vert:Double,
       amp:Double, period:Double):ImageTrans[T] = {
   (image:Image[T]) => {
      (col:Double, row:Double) => {
         val scaleAmount = waveIntensityOrigin(phase,
                              vert, amp, period)(col, row)
         scaleOrigin(scaleAmount)(image)(col, row)
      }
   }
}
```

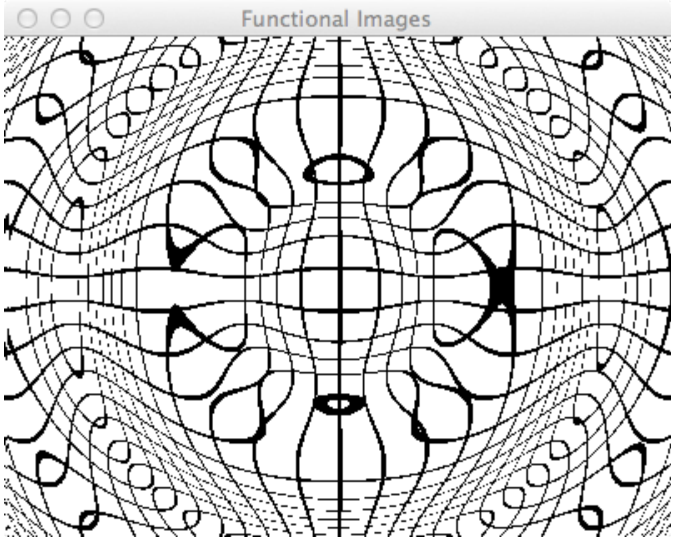
Fancy animation

```
def waveScale[T](phase: Double, vert:Double, amp:Double,
                 period:Double,
                 col:Double, row:Double):ImageTrans[T] = {

   aboutPoint(waveScaleOrigin(phase, vert, amp, period),
      col, row)
}
```

Bending the grid image over the wave

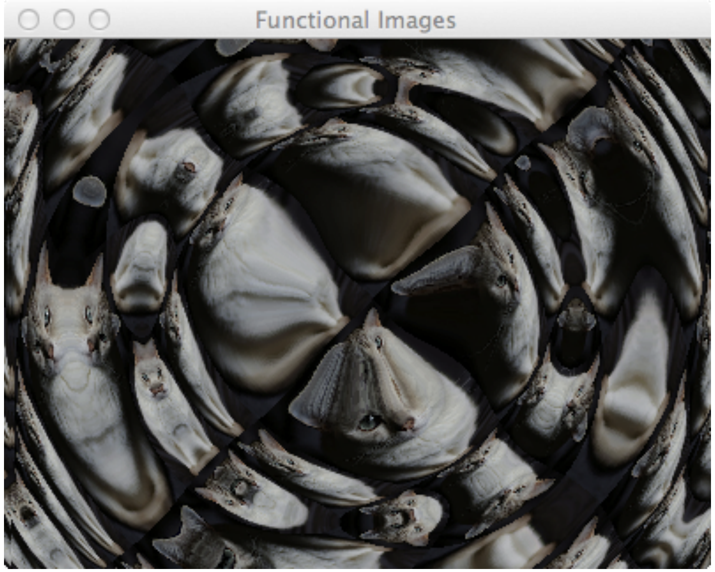
```
def waveGridAnimation(time:Double):Image[Boolean] = {
   waveScale(time * 2, 1, 0.3, 100, 200, 150)(grid(20, 2))
}
new Animate(400, 300, waveGridAnimation).show()
```



above is a still shot from the animation

Bending a bitmap over the wave

```
def waveBitmapAnimation(time:Double):Image[Color] = {
   waveScale(time * 2, 2, 0.8, 100, 200, 150)
            (scaledRotatedBitmap)
}
new Animate(400, 300, waveBitmapAnimation).show()
```



above is a still shot from the animation

Conclusion

- Code is on github: https://github.com/bjpop/scala-fungraph
- What would it take to add interaction to animations? Conal Elliot: Functional Reactive Animation (1997), and later Functional Reactive Programming.
- Should be straightforward to parallelize; the trick is getting the granularity right.
- But difficult to “share” computations between pixels.