6 Functional images

This practical expores one way of generating bitmap images of the form that we were rendering in Practical 4. But rather than cartoon cats, or lettering, or photographs, the images will be more geometric in nature.

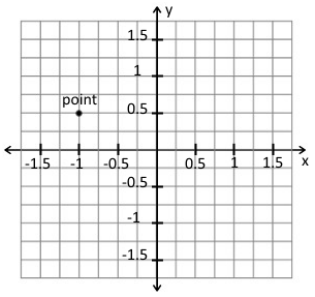
The crucial observation is that a bitmap image assigns a shade—a bit, a grey value, or a colour value—to each pixel in the image; that is, it is a function from pixels to shades. Traditional image formats like GIF, JPG, and PNM, and the list-of-lists representation we used in Practical 4, represent this function indirectly, by explicitly specifying its value for every possible element of the (finite!) domain of the function. But functional programming makes available another approach: we can represent the function from pixels to shades *directly*, by specifying instead the method for computing the shade of each pixel. That is, an image is a *function*, and operations that manipulate images are *higher-order functions*, functions that work on other functions.

This approach to image representation is based on the Pan language designed by Conal Elliott. Elliott wrote a chapter *Functional Images* for a book *The Fun of Programming* that I co-edited; see `http://conal.net/papers/functional-images/`, and especially browse through the gallery of images for inspiration. Pan arose out of earlier work of Elliott with Paul Hudak on *Functional Reactive Animation*, in which animations too are represented as functions—for example, from time to image. Elliott and Hudak's paper of that title from ICFP 1997 won the 'most influential paper' award ten years later for its impact on the field. That line of work has subsequently evolved into *Functional Reactive Programming*, which generalises from animated images to other time-dependent behaviours, such as robotics and graphical user interfaces; some of that work is covered in Hudak's textbook *The Haskell School of Expression*.

The essential idea is that a two-dimensional image is a function from points in the 2D plane to shades. We represent the 2D plane as complex numbers, using the library *Data.Complex*:

```
type CF = Complex Float

point :: CF
point = (-1.0) :+ 0.5
```

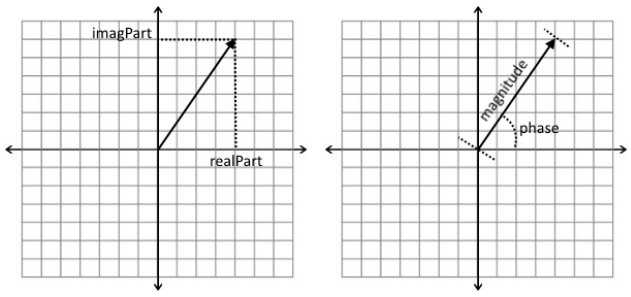


Complex numbers are of course an instance of Haskell's *Num* type class, so they come with an assortment of useful numeric functions such as addition and multiplication; we could just have used pairs, but then we would have had to define addition and multiplication ourselves. The constructor :+ constructs a complex number $x :+ y = x + i\ y$ from its 'real' and 'imaginary' parts x and y, where $i = \sqrt{-1}$. But you don't need to worry about imaginary numbers if you haven't encountered them before; just think of (x, y) as a point in the plane. There are two functions to extract the Cartesian coordinates of a point:

$$realPart, imagPart :: CF \to Float$$

and two more to extract the polar coordinates:

$$magnitude, phase :: CF \to Float$$



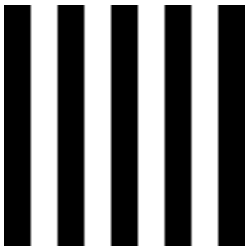
An image with shades of type c is simply a function from CF to c:

type $Image\ c = CF \to c$

For example, the following image has unit-width columns, alternating between black and white: a pixel is black (*True*) if the integer part of its x-coordinate is even, and white (*False*) if it is odd.

```
cols :: Image Bool
cols = even ∘ floor ∘ realPart
```

Note that an image is infinite in width and height, and also infinite in precision; when rendering it, we have to specify a finite window into the infinite whole, and specify at what resolution to sample the image within that window. So if we sample the image *cols* over 90×90 points evenly spaced between $0.05 :+ 0.05$ and $8.95 :+ 8.95$, we get the following bitmap:



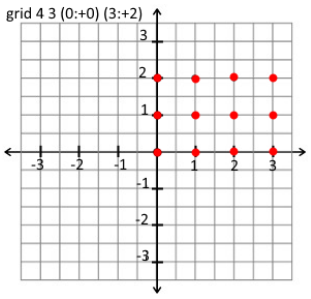
(We've carefully chosen the coordinates to avoid whole numbers, to prevent possible rounding issues with *Float*s.)

1. Define a function

   $$grid :: Int \to Int \to CF \to CF \to Grid\ CF$$

   that takes two integers m, n and two complex numbers p, q and constructs an $m \times n$ grid of complex numbers, evenly spaced between p and q. For example,

   $$\begin{aligned} &grid\ 4\ 3\ (0 :+ 0)\ (3 :+ 2) \\ &\quad = [\,[\,0.0 :+ 2.0, 1.0 :+ 2.0, 2.0 :+ 2.0, 3.0 :+ 2.0\,], \\ &\qquad [\,0.0 :+ 1.0, 1.0 :+ 1.0, 2.0 :+ 1.0, 3.0 :+ 1.0\,], \\ &\qquad [\,0.0 :+ 0.0, 1.0 :+ 0.0, 2.0 :+ 0.0, 3.0 :+ 0.0\,]\,] \end{aligned}$$

   Note that the y-coordinates are in reverse order, so that if the given points are the bottom-left and top-right corners then the

first row is the top row. (Hint: try a few concrete examples on paper, such as *grid* 5 3 (1 :+ 0) (3 :+ 4) Can you find the pattern? List comprehensions might be useful!)

```
grid c r (pr :+ pi) (qr :+ qi)
  = [[zr :+ zi | zr ← for c pr qr] | zi ← for r qi pi]
    where
      for n a b = take n [a, a + d..]
        where d = (b − a) / fromIntegral (n − 1)
```

Note that we generate an infinite list $[a, a+d..]$ and then *take* the requisite number of elements from this list. What might go wrong if you try to generate the finite list $[a, a+d..a+(n-1)*d]$ directly?

2. Define a function

   ```
   sample :: Grid CF → Image c → Grid c
   ```

   that takes a grid of sample points (like that returned by *grid* above) and a continuous image, and samples the image at each of the given grid points. For example,

   ```
   charRender (bwCharView
     (sample (grid 7 7 (0.5 :+ 0.5) (6.5 :+ 6.5)) cols))
   ```

   should yield the following output:

   ```
   * * * *
   * * * *
   * * * *
   * * * *
   * * * *
   * * * *
   * * * *
   ```

   and

   ```
   pbmRender "test.pbm"
     (sample (grid 90 90 (0.05 :+ 0.05) (8.95 :+ 8.95)) cols)
   ```

   the image above.

```
sample points image = map (map image) points
```

3. Define an image

   rows :: *Image Bool*

   consisting of alternating black and white horizontal unit-height rows.

```
rows = even ∘ floor ∘ imagPart
```

4. Define an image

   chequer :: *Image Bool*

   consisting of a chequerboard of unit-size squares. Can you reuse the definitions of *cols* and *rows*?

Here's one solution:

```
chequer z = even (floor (realPart z) + floor (imagPart z))
```

but it is possible to do it more simply:

```
chequer z = (cols z == rows z)
```

5. Define an image

   rings :: *Image Bool*

   of alternating black and white unit-width rings about the origin. For example,

```
pbmRender "test.pbm"
  (sample (grid 100 100 (−(10 :+ 10)) (10 :+ 10)) rings)
```

   should yield

(Now we need not worry so much about rounding errors, so we just sample at integral grid points.)

```
rings = even ∘ floor ∘ magnitude
```

6. Define an image

   ```
   wedges :: Int → Image Bool
   ```

   so that *wedges n* consists of $2n$ alternating black and white wedges, radiating from the origin; for example, *wedges* 12 yields

```
wedges n = even ∘ floor ∘ (∗(fromIntegral n / pi)) ∘ phase
```

7. Can you provide definitions of *rings* and *wedges* in terms of *cols* and *rows*?

The trick is to transform from Cartesian coordinates to polar coordinates:

```
toPolar :: CF → CF
toPolar z = magnitude z :+ phase z

rings'      = cols ∘ toPolar
wedges' n = rows ∘ (∗(fromIntegral n / pi)) ∘ toPolar
```

8. Define a polar chequerboard image with an even number of 'wedges'

   ```
   polarChequer :: Int → Image Bool
   ```

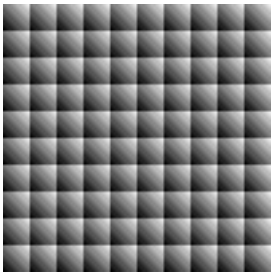
   like that at the start of Practical 4.

```
polarChequer n z = chequer (r :+ (fromIntegral n ∗ theta / pi))
  where r :+ theta = toPolar z
```

9. All the images so far have been bitmaps; but the *Image* type is polymorphic in the shade type, so that's not required. Define a greyscale image

   ```
   shadedChequer :: Image Float
   ```

   consisting of a chequerboard in which each square is gradually shaded from black (shade 0.0) in the bottom left corner to white (1.0) in the top right:



```
shadedChequer (x :+ y) = (fracPt x + fracPt y) / 2
  where fracPt x = (x − fromIntegral (floor x))
```

10. Try to replicate some of the images in Conal Elliott's gallery at `http://conal.net/Pan/Gallery/`.