Andrew Cooke

Pancito 2.2

Introduction

Pancito 2.2 Another minor release. Apart from random, undocumented changes (no-one uses this, as far as I know, apart from me), the I/O routines have been largely rewritten and some type classes introduced. Changes have been driven by practical use of the system – I have been working on several images and hope, eventually, to have a gallery on the 'net.

Important features of Pancito include:

- Pure, simple, functional Haskell. Images are purely functional, only the IO code requires a Monad.
- Useful practical features – image tiling, progress bars. I suspect this package has been used more than any other functional image package for 'artistic' work.
- Open, extensible framework. I'm moving towards parametric functions and modular code in many areas (eg the Image' a type, and the modular output routines).
- Support for reading files from disk.

Pancito 2.1 This is a minor release of Pancito that has been extended to allow images to be read from files. It is otherwise identical to 2.0.

Pancito 1 The original Pancito was inspired by Pan[1] — I tried to provide basic support for functional images on non-Win32 platforms. This boiled down to:

- A collection of types for the basic components (Images, Colours, Transforms etc)
- Basic support for these types (mainly Colours)
- Output to a disk file

In retrospect this suffered from several problems (I'm talking about Pancito here; I have not used Pan enough to know anything more than that it is a lot faster):

- Speed. It's slow. Sorry, but this doesn't bother me much - I can develop ideas with small images/windows.
- Clunky interfaces. For example: Colour was fixed as an RGBA representation; sometimes I wanted to experiments with HSV. And Point was in rectangular coordinates (unless it wasn't — the compiler couldn't check).

[1] http://www.conal.net/pan

- No higher order functions. Apart from functional composition I didn't give much thought about how to combine different components to assemble images.
- The only components that varied with location were images. Using Pancito later, I had to write extra code for (boolean) masks.
- Poor support for (efficient) transformations that require several points in an image (for example, blurring). This is tricky within the functional image approach and I don't have any good ideas about how to improve this yet.

Pancito 2 is going to fix some of these problems, I hope — it's also an opportunity for me to think a bit more about software engineering (rather than just throwing some code together) in Haskell. If anyone does use both, I'd like to know which is easier to use. Cheers.

Version This is version 2.2. It is partially backwards compatible with Pancito 2.1. It is *not* compatible with Pancito 1.

Last altered: 10 February 2004.

Haskell This document, which describes the Pancito module, includes all the source code. Haskell compilers (at least, hugs, ghc and nhc98) should be able to compile Pancito directly from the document you are reading.

Point

Header This module implements Point as an ADT. This has some disadvantages — you can't pattern match to get the individual components — but it provides a safe way of switching between polar and rectangular coordinates and it lets me introduce checks for value boundaries.

And in 2.2, Point' is now a type class which Point implements. The Pancito package has support for writing the general Point' class, but the final output routines are for Point (since that is probably what you want and, if you don't, you know enough to pull in the general functions). To see Point' in use, check the Flux module.

```
module Point (
  Point', x, y, d, t, optimizePolar, optimizeCartesian,
  setXy, setDt, setX, setY, setD, setT,
  Point, cartesian, polar,
  xy, dt, Transform', Transform, shift, unshift, scale, unscale,
  shift', unshift', rotate, unrotate, expand,
  Window', Window, origin, unit, nUnit, square01, square11, mapWindow,
  Region', Region, contains, contains', winRegion,
  rot90, rot180, rot270, flipX, flipY, pixelate11,
  add', sub',
  trapezium, trapezium', pincushion, Line, nearest
) where

import Common
```

Constructor, Classes

The code is based on an example posted to the Haskell mailing list by Tom Pledger. By using two different representations we can avoid unnecessary conversions (if the user is explicit in writing conversions) but still support both representations at any point.

```
class (Eq p, Show p) ⇒ Point' p where
  x :: p → Double
  y :: p → Double
  d :: p → Double
  t :: p → Double
  optimizeCartesian :: p → p
  optimizePolar :: p → p
  setXy :: Double → Double → p → p
  setDt :: Double → Double → p → p
  setX :: Double → p → p
  setX x p = setXy x (y p) p
  setY :: Double → p → p
  setY y p = setXy (x p) y p
  setD :: Double → p → p
  setD d p = setDt d (t p) p
  setT :: Double → p → p
  setT t p = setDt (d p) t p

data Point = Cartesian Double Double
           | Polar     Double Double

instance Eq Point where
  p1 == p2 = equalPoint p1 p2

instance Point' Point where
  x = x'
  y = y'
  d = d'
  t = t'
  optimizeCartesian = optimizeCartesian'
  optimizePolar = optimizePolar'
  setXy x y _ = cartesian x y
  setDt d t _ = polar d t

equalPoint (Cartesian x1 y1) (Cartesian x2 y2) = x1 == x2 && y1 == y2
equalPoint (Polar d1 t1) (Polar d2 t2) = d1 == d2 && t1 == t2
equalPoint p1 p2 = equalPoint p1' p2'
  where
    p1' = optimizeCartesian p1
    p2' = optimizeCartesian p2

instance Show Point where
  show p = showPoint p

showPoint (Cartesian x y) = "(" ++ show x ++ "," ++ show y ++ ")"
showPoint (Polar d t) = "p(" ++ show d ++ "," ++ show t ++ ")"
```

Proxy Constructors

The constructor is hidden — these should be used instead. Note that explicit conversion is optional (it's provided to allow code to be optimized, hence the name) and isn't as important as for Colours because conversino is no less expensive than using d and t once each (so optimizing is only worthwhile if you will repeatedly use polar values from the same Point).

```
cartesian, polar :: Double → Double → Point
```

```
cartesian x y = Cartesian x y
polar d t     = Polar d (roll2pi t)

optimizeCartesian', optimizePolar' :: Point → Point
optimizeCartesian' p = case p of
                         (Polar d t)     → cartesian (x p) (y p)
                         (Cartesian x y) → p
optimizePolar' p  = case p of
                      (Polar d t)     → p
                      (Cartesian x y) → polar (d p) (t p)
```

Access The following operators pull values out from the ADT. I've used d rather than r for the polar radial distance so that r can be used for the red component of colours.

```
x', y', d', t' :: Point → Double
x' (Cartesian x y) = x
x' (Polar d t)     = d * cos t
y' (Cartesian x y) = y
y' (Polar d t)     = d * sin t
d' (Cartesian x y) = sqrt (x*x + y*y)
d' (Polar d t)     = d
t' (Cartesian x y) = atan2 y x
t' (Polar d t)     = t

xy, dt :: Point' p ⇒ p → (Double, Double)
xy p = (x p', y p')
  where
    p' = optimizeCartesian p
dt p = (d p', t p')
  where
    p' = optimizePolar p
```

Modifying Points Remember that these alter coordinates before being passed to the image, and so are counterintuitive. For example, scaling the coordinates expands to include more of the image — more of the image is displayed in the same area, so it appears "smaller".

```
type Transform' p = p → p
type Transform = Transform' Point

shift, unshift, scale, unscale :: Point' p ⇒ Double → Double → Transform' p
shift dx dy p = setXy (x p + dx) (y p + dy) p
unshift dx dy p = setXy (x p - dx) (y p - dy) p
scale fx fy p = setXy (fx * x p) (fy * y p) p
unscale fx fy p = setXy ((1.0 / fx) * x p) ((1.0 / fy) * y p) p

shift', unshift' :: Point' p ⇒ p → Transform' p
shift' p = shift (x p) (y p)
unshift' p = unshift (x p) (y p)

rotate, unrotate, expand :: Point' p ⇒ Double → Transform' p
rotate theta p = setDt (d p) (t p + theta) p
unrotate theta p = setDt (d p) (t p - theta) p
expand k p = setD (k * d p) p
```

Windows Often it is necessary to convert from one ractangular region to another.

```
type Window' p = (p, p)
```

```
type Window = Window' Point

origin, unit, nUnit :: Point
origin = cartesian 0.0 0.0
unit = cartesian 1.0 1.0
nUnit = cartesian (−1.0) (−1.0)

square01, square11 :: Window
square01 = (origin, unit)
square11 = (nUnit, unit)

mapWindow :: Point' p ⇒ Window' p → Window' p → Transform' p
mapWindow (toBL, toTR) (fromBL, fromTR) =
    shift dx dy ∘ scale fx fy
  where
    fx = (x toTR − x toBL) / (x fromTR − x fromBL)
    fy = (y toTR − y toBL) / (y fromTR − y fromBL)
    dx = x toBL − x fromBL ∗ fx
    dy = y toBL − y fromBL ∗ fy
```

Windows are the simplest example of a more general idea — describing a specific region of an image. The Window type is convenient for ractangles, but for more complex shapes we need functional Regions.

```
type Region' p = p → Bool
type Region = Region' Point

contains :: Point' p ⇒ Region' p → p → Bool
contains r p = r p  −− syntactic sugar

contains' :: Point' p ⇒ Window' p → p → Bool
contains' = contains ∘ winRegion

winRegion :: Point' p ⇒ Window' p → Region' p
winRegion (bl, tr) p =
  (x bl − x p) ∗ (x p − x tr) ≥ 0 &&
    (y bl − y p) ∗ (y p − y tr) ≥ 0
```

Various Tools Some simple flips and rotations.

```
rot90, rot180, rot270, flipX, flipY :: Point' p ⇒ Transform' p
rot90 p = setXy (−(y p)) (x p) p
rot180 p = setXy (y p) (x p) p
rot270 p = setXy (y p) (−(x p)) p
flipX p = setY (−1 ∗ y p) p
flipY p = setX (−1 ∗ x p) p
```

Transform the window ((n,m) (n+1,m+1)) to ((-1,-1) (1,1)). So each unit square is mapped to the same square11.

```
pixelate11 :: Point' p ⇒ Transform' p
pixelate11 p = setXy (f x) (f y) p
  where
    p' = optimizeCartesian p
    f xy = ((xy p') − fromIntegral (floor (xy p'))) ∗ 2.0 − 1.0
```

One day these might use overload arithmetic operators, but it's a lot of fuss.

```
add', sub' :: Point' p ⇒ p → p → p
add' p1 p2 = setXy (x p1 + x p2) (y p1 + y p2) p1
sub' p1 p2 = setXy (x p1 − x p2) (y p1 − y p2) p2
```

And the adjustments on a TV.

```
trapezium, trapezium' :: Point' p ⇒ Double → Transform' p
trapezium k p = setY (y'*(1.0+k*x')) p
  where
    (x', y') = (x p, y p)
trapezium' k p = setX (x'*(1.0+k*y')) p
  where
    (x', y') = (x p, y p)

pincushion :: Point' p ⇒ Double → Transform' p
pincushion k p = setD ((d p) ** k) p
```

The point on a line nearest a given point. We define a line as a point and a gradient.

```
type Line p = (p, Double, Double)

nearest :: Point' p ⇒ Line p → p → p
nearest (o, dx, dy) p = setXy x' y' p
  where
    (ox, oy) = xy o
    (px, py) = xy p
    x' = (px*dx*dx+(py–oy)*dx*dy+ox*dy*dy)/(dx*dx+dy*dy)
    y' = (oy*dx*dx+(px–ox)*dx*dy+py*dy*dy)/(dx*dx+dy*dy)
```

Colour

Header

This module implements Colour as an ADT, just like Point. It also defines some useful operations. The original Pancito had a whole pile of Colour functions that I never used; this one doesn't. Other changes include HSV support and pre-multiplied alpha.

```
module Colour (
  Colour, rgba, hsva, optimizeHsva, optimizeRgba, tiny,
  r, g, b, a, h, s, v,
  setR, setG, setB, setA, setH, setS, setV,
  Tint', Tint, red, green, blue, black, white,
  opacity, transparent, interp, lighten, darken, gamma,
  cyan, magenta, yellow, grey,
  rotateHue, saturate, brighten, brighten',
  overlay, combine, add, sub, average', average, cMap,
  ranColour, ranColour'
) where

import Common
import Random
```

Constructor, Classes

Again, by using two different representations we can avoid unnecessary conversions (if the user is explicit in writing conversions) but still support both representations at any point. We can also check that components are within accepted value ranges.

However, there are some disadvantages to using two different representations: some operations depend on the underlying type. In particular, adding two colours gives different results for HSVA and RGBA encoded values. This is ugly, but I'm not sure what the best solution is. At the moment, you are free to force the type using the two optimize functions. An alternative would be to provide different functions for the two operations.

```
data Colour = RGBA Double Double Double Double
            | HSVA Double Double Double Double

instance Eq Colour where
  p1 == p2 = equalColour p1 p2

equalColour (RGBA r1 g1 b1 a1) (RGBA r2 g2 b2 a2) =
  r1 == r2 && g1 == g2 && b1 == b2 && a1 == a2
equalColour (HSVA h1 s1 v1 a1) (HSVA h2 s2 v2 a2) =
  h1 == h2 && s1 == s2 && v1 == v2 && a1 == a2
equalColour c1 c2 = equalColour c1' c2'
  where
    c1' = optimizeRgba c1
    c2' = optimizeRgba c2

instance Show Colour where
  show p = showColour p

showColour (RGBA r g b a) =
  "rgb(" ++ show r ++ "," ++ show g ++ "," ++
    show b ++ "," ++ show a ++ ")"
showColour (HSVA h s v a) =
  "hsv(" ++ show h ++ "," ++ show s ++ "," ++
    show v ++ "," ++ show a ++ ")"
```

Proxy Constructors

The constructor is hidden — these should be used instead. They enforce value ranges (including pre–multiplied alpha) which are assumed when values are accessed later.

```
rgba, hsva :: Double → Double → Double → Double → Colour
rgba r g b a = RGBA (clip0 a' r) (clip0 a' g) (clip0 a' b) a'
  where a' = (clip01 a)
hsva h s v a = HSVA (roll2pi h) (clip01 s) (clip0 a' v) a'
  where a' = (clip01 a)
```

Conversion

Again, note that explicit conversion is optional, but improves efficiency.

The conversions here are lifted from several posts on the internet (search for "RGB HSV convert"), translated from C or pseudocode to Haskell. I hope an optimizing compiler can simplify the logic in the comparisons (I could do it by hand, but then it would even less readable).

```
tiny = 0.000001

mkHSVA max min n x y a =
    hsva (rad * (n + xy)) s max a
  where
    d = max - min
    s = if max > tiny then d / max else 0
    xy = if d > tiny then (x - y) / d else 0
    rad = pi / 3.0

rgbaToHsva (RGBA r g b a)
  | abs (r - g) < tiny && abs (g - b) < tiny &&
    abs (b - r) < tiny = hsva 0 0 r a
  | order b g r = mkHSVA r b 0 g b a
  | order g b r = mkHSVA r g 0 g b a
  | order r b g = mkHSVA g r 2 b r a
  | order b r g = mkHSVA g b 2 b r a
  | order g r b = mkHSVA b g 4 r g a
```

```
  | order r g b = mkHSVA b r 4 r g a
  where
    order x y z = x ≤ y && y ≤ z

optimizeHsva :: Colour → Colour
optimizeHsva c = case c of
                   (RGBA r g b a) → rgbaToHsva c
                   (HSVA h s v a) → c

mkRgba i p q t v a
  | i == 0    = rgba v t p a
  | i == 1    = rgba q v p a
  | i == 2    = rgba p v t a
  | i == 3    = rgba p q v a
  | i == 4    = rgba t p v a
  | i == 5    = rgba v p q a
  | otherwise = error ("case " ++ show i ++ " for " ++ show p ++ ", "
                       ++ show q ++ ", " ++ show t ++ ", "
                       ++ show v ++ ", " ++ show a ++ " in mkRgba")

hsvaToRgba (HSVA h s v a)
  | s == 0.0  = rgba v v v a
  | otherwise = mkRgba i p q t v a
  where
    h' = h / rad
    rad = pi / 3.0
    i = floor h'
    f = h' - fromIntegral i
    p = v * (1 - s)
    q = v * (1 - s * f)
    t = v * (1 - s * (1 - f))

optimizeRgba :: Colour → Colour
optimizeRgba c = case c of
                   (RGBA r g b a) → c
                   (HSVA h s v a) → hsvaToRgba c
```

Access The following operators pull values out from the ADT.

```
r, g, b, a, h, s, v :: Colour → Double
r c = case c of
        (RGBA r' g b a) → r'
        (HSVA h s v a)  → r (hsvaToRgba c)
g c = case c of
        (RGBA r g' b a) → g'
        (HSVA h s v a)  → g (hsvaToRgba c)
b c = case c of
        (RGBA r g b' a) → b'
        (HSVA h s v a)  → b (hsvaToRgba c)
a c = case c of
        (RGBA r g b a') → a'
        (HSVA h s v a') → a'
h c = case c of
        (RGBA r g b a)  → h (rgbaToHsva c)
        (HSVA h' s v a) → h'
s c = case c of
        (RGBA r g b a)  → s (rgbaToHsva c)
        (HSVA h s' v a) → s'
v c = case c of
        (RGBA r g b a)  → v (rgbaToHsva c)
        (HSVA h s v' a) → v'
```

Modification Probably not very useful if alpha used (pre-multiplied).

```
setR, setG, setB, setA, setH, setS, setV :: Double → Colour → Colour
setR r c = rgba r (g c') (b c') (a c')
  where c' = optimizeRgba c
setG g c = rgba (r c') g (b c') (a c')
  where c' = optimizeRgba c
setB b c = rgba (r c') (g c') b (a c')
  where c' = optimizeRgba c
setA a c = rgba (r c') (g c') (b c') a
  where c' = optimizeRgba c
setH h c = hsva h (s c') (v c') (a c')
  where c' = optimizeHsva c
setS s c = hsva (h c') s (v c') (a c')
  where c' = optimizeHsva c
setV v c = hsva (h c') (s c') v (a c')
  where c' = optimizeHsva c
```

Utilities First, a few colours and simple tools. I've not provided anything to force a single component to a value (but see combining colours later). The Tint type will be discussed later when bring everything together into a pipeline.

Note that, unlike Pancito 1, alpha is pre-multiplied (thanks to Conal Elliott for point me to Alvy Ray Smith's work[2]).

```
type Tint' a b = a → b
type Tint = Tint' Colour Colour

red, green, blue, black, white :: Colour
red = rgba 1 0 0 1
green = rgba 0 1 0 1
blue = rgba 0 0 1 1
white = rgba 1 1 1 1
black = rgba 0 0 0 1

opacity :: Double → Tint
opacity x (RGBA r g b a) = rgba r' g' b' x
  where
    r' = rescale r a x
    g' = rescale g a x
    b' = rescale b a x
opacity x (HSVA h s v a) = hsva h s' v' x
  where
    s' = rescale s a x
    v' = rescale v a x

rescale x old new | old < tiny  = 0  -- transparent is black?
                  | otherwise   = x * new / old

transparent = opacity 0 black
```

Colours can be interpolated (this preserves alpha from the *second* colour, allowing functions like lighten and darken that leave transparency unchanged to be defined using currying):

```
interp :: Double → Colour → Tint
interp frac c1 c2 =
    rgba (intp r) (intp g) (intp b) a2
```

[2] http://www.alvyray.com — Memo "Image Compositing Fundamentals".

```
  where
    intp f = (\x → x + frac * ((f c2') - x)) (f c1')
    a2 = a c2
    c1' = opacity a2 $ optimizeRgba c1
    c2' = optimizeRgba c2

lighten, darken :: Double → Tint
lighten frac = interp (1 - frac) white
darken frac = interp (1 - frac) black

cyan, magenta, yellow, grey :: Colour
cyan = interp 0.5 blue green
magenta = interp 0.5 red blue
yellow = interp 0.5 green red  —— 0.6 looks better on my screen
grey = interp 0.5 black white
```

The equivalent of interpolation when thinking is HSVA space is a bunch of separate actions:

```
rotateHue, saturate, brighten, brighten' :: Double → Tint
rotateHue theta c = hsva (h c' + theta) (s c') (v c') (a c)
  where
    c' = optimizeHsva c
saturate x c = hsva (h c') (s c' + x) (v c') (a c)
  where
    c' = optimizeHsva c
brighten x c = hsva (h c') (s c') (v c' + x) (a c)
  where
    c' = optimizeHsva c
brighten' x c = hsva (h c') (s c') (v c' * x) (a c)
  where
    c' = optimizeHsva c
```

Then there's gamma correction.

```
gamma :: Double → Tint
gamma x c = rgba ((r c')**x) ((g c')**x) ((b c')**x) ((a c')**x)
  where
    c' = optimizeRgba c
```

Combining Colours

The canonical way to combine colours is by overlaying and letting the alpha channel do its work. In Pancito 1, "underlay" had reversed arguments for use with foldl. I now understand foldr is the fundamental fold, so am back with "overlay" (the first argument is laid over the second).

```
overlay :: Colour → Colour → Colour
overlay c1 c2 = rgba (ov r) (ov g) (ov b) (ov a)
  where
    c1' = optimizeRgba c1
    c2' = optimizeRgba c2
    beta = a c1
    ov clr = (clr c1') + (1 - beta) * (clr c2')
```

If this isn't what you want (it often isn't when you're dealing with pure colour images, as it loses saturation) the following might help (alpha is set to 1 to avoid limiting when pre–multiplication is enforced).

```
combine :: (Double → Double → Double) →
             Colour → Colour → Colour
combine f (HSVA h1 s1 v1 a1) (HSVA h2 s2 v2 a2) =
    hsva (f h1 h2) (f s1 s2) (f v1 v2) 1
```

```
combine f (RGBA r1 g1 b1 a1) (RGBA r2 g2 b2 a2) =
    rgba (f r1 r2) (f g1 g2) (f b1 b2) 1
combine f c1 c2 = combine f c1' c2'
  where
    c1' = optimizeRgba c1
    c2' = optimizeRgba c2

add, sub :: Colour → Colour → Colour
add = combine (+)
sub = combine (−)
```

An average (RGBA) of a list of colours is useful sometimes.

```
average' :: Colour → (Colour, Int) → (Colour, Int)
average' c1 (c2, n) = (clr, n')
  where
    n' = n + 1
    clr = rgba (av r) (av g) (av b) (av a)
    c1' = optimizeRgba c1
    c2' = optimizeRgba c2
    k1 = 1.0 / fromIntegral n'
    k2 = fromIntegral n / fromIntegral n'
    av f = (k1 * f c1') + (k2 * f c2')

average :: [Colour] → Colour
average = fst ∘ foldr average' (black, 0)
```

It can also be convenient to apply a function to each component of a colour.

```
cMap :: (Double → Double) → Colour → Colour
cMap f (HSVA h1 s1 v1 a1) = hsva (f h1) (f s1) (f v1) 1
cMap f (RGBA r1 g1 b1 a1) = rgba (f r1) (f g1) (f b1) 1


ranColour' :: StdGen → [Colour]
ranColour' ran = ranColour'' $ randoms ran

ranColour'' :: [Double] → [Colour]
ranColour'' (r':g':b':rgb) = (rgba r' g' b' 1.0):(ranColour'' rgb)

ranColour :: Int → [Colour]
ranColour n = ranColour' $ mkStdGen n
```

Pancito2

Header This builds on the Colours and Points defined earlier to provide support for generating and writing Images.

```
module Pancito2 (
  Image'', Image', Image, flat, Merge, foldc, over, avg,
  boxToWindow, dimToWindow, dimToBox,
  VTint', VTint, o, Filter', Filter, mkFilter, pixelStream, pixelToPoint,
  openPpm, writePpm, closePpm, ppmBase, tileBox,
  MkBox, MkImage, PxWrite, MkPixels, idBox, idImage, idWrite, idPixels,
  ppm, ppmAlias, ppmAlias', tile, tileAlias, tileAlias',
  monWrite, monPixels, countBox,
  ppm_, ppmAlias_, ppmAlias'_, tile_, tileAlias_, tileAlias'_
) where
```

```
import Point
import Colour
import IO
import Control.Monad

import System.IO.Unsafe
```

Image

Pancito considers images to be functions. Give an image a point and it gives you a colour:

```
type Image'' p a = p → a
type Image' p = Image'' p Colour
type Image = Image' Point

flat :: Colour → Image' p
flat c _ = c
```

Simple Image Pipelines

It's useful to think of a functional image as a pipeline (a flow from right to left matches Haskell's syntax when using function application or composition). To find the colour at a particular point in an image we pop the Point into the right of the pipeline. The Point can then pass through Transforms (defined in Point — these can alter the Point coordinates, distorting the image) before entering an Image. The Image converts the Point to a Colour. The Colour then continues down the pipe to the left, possibly passing through Tints that alter it further, before finally appearing to us.

Simple Tints can be made from functions defined in Colour (brighten or saturate, for example); Point has several Transforms.

The description above is a very simple pipeline. Now we will extend them to include several pipes in parallel and Tints that can vary with position.

Parallel Images

A list of images can be generated in parallel and combind in some way. The simplest example of combination is by overlaying.

```
type Merge' p a = [Image'' p a] → Image'' p a
type Merge p = Merge' p Colour

foldc :: (a → b → b) → b → [Image'' p a] → Image'' p b
foldc f v ims p = foldr (\im v' → f (im p) v') v ims

over :: Colour → Merge p
over = foldc overlay

avg :: Merge p
avg ims p = fst $ foldc average' (black, 0) ims p
```

That code is fairly dense, so some examples will probably help clear things up:

```
im1, im2, imR, imG :: Image' p
imR = flat red
imG = flat green
im1 = over white [imR, imG]
im2 = foldc add black [imR, imG]
```

Here im1 is white overlaid with a red and then a green image (and since green is opaque, the result will be green). Im2 is formed by adding the individual colour components and so will be yellow.

Varying Tints The type system described so far doesn't support Tints that vary with position (within a pipeline defined purely using functional composition a Tint does not receive a Point).

We can always work round this:

```
{-
mkTint :: Point → Tint

im1, im2 :: Image'
im1 p = tint ∘ im2 p
  where tint = mkTint p
-}
```

but it's pretty ugly. Instead, we can replace the "." of functional composition with a different operator, "o" that pulls a Point from earlier in the pipeline.

```
type VTint' p = p → Tint
type VTint = VTint' Point

o :: VTint' p → Image' p → Image' p
o mkTint im p = mkTint p $ im p
infixr 8 `o`
```

For example,

```
{-
brightPos :: VTint
brightPos p = if (x p > 0) then brighten 0.5 else brighten (-0.5)

im1, im2, imR :: Image'
imR = flat red
im1 = brightPos `o` imR ∘ shift 1 2
im2 = (brightPos `o` imR) ∘ shift 1 2
-}
```

Note that because "o" binds less strongly that "." the coordinate received by the tint in im1 above will not be shifted. This behaviour can be altered using parentheses, as in im2.

Filter A still more general operation is Filter.

```
type Filter' p = Image' p → Image' p
type Filter = Filter' Point
```

While this could be any code, a simple way of building filters is by assembling a Tint and a Transform.

```
mkFilter :: Tint → Transform' p → Filter' p
mkFilter tint trans im = tint ∘ im ∘ trans
```

Output Yet again, the code here has changed. It was an embarassing mess before. Hopefully this will be much clearer (although more succinct).

Pixels and Points

An image is a function defined over a fixed range of points – the image has a fixed domain. However, we may want to "realise" the image at a variety of different resolutions/sizes. So the number of pixels used to generate the image may vary.

It's natural (and useful when tiling, for example) to identify pixels with Points at integer coordinates, but this raises a problem: as we change the number of pixels, we alter the range of values over which we evaluate the function.

The functions below attempt to protect the user from this. Images are assumed to be defined over a certain window, which is supplied to the output routines. It is the responsibility of the output routines to generate the approrpiate transforms from pixel coordinates to the coordinates defined by the image window.

Also, for various rather subtle practical reasons, it simplifies the code if the centres of pixels are at 0.5, 1.5, etc. Since each pixel has unit dimension this means that the bottom left corner of the image (bottom left corner of the bottom left pixel) is (0,0) and the top right corner is (nx, ny).

The type Box is used internally to define the range of pixel values required.

```
type Box = ((Int, Int), (Int, Int))
```

ppm Format

The ppm image format is simple, text based, verbose, and has no support for transparent images, but it does support arbitrary colour depth and can be converted to a wide variety of other formats using the netpbm and pbmplus packages (for CMYK encoded tiff files, see my own pnmtocmyktiff).

A line in a ppm file can contain a maximum of 70 characters...

```
lineLimit :: Int
lineLimit = 69 -- allow for \n

putChunk :: Handle → Int → String → IO Int
putChunk h soFar str
    | soFar == 0       = do hPutStr h str
                            return len
    | tot ≥ lineLimit = do hPutChar h '\n'
                            hPutStr h str
                            return len
    | otherwise        = do hPutChar h ' '
                            hPutStr h str
                            return tot
  where len = length str
        tot = soFar + len + 1
```

...and begins with the following header:

```
headerPpm :: Handle → Box → Int → IO (Handle)
headerPpm h ((xlo, ylo), (xhi, yhi)) n = do hPutStrLn h "P3"
                                            hPutStrLn h (show x)
                                            hPutStrLn h (show y)
                                            hPutStr h (show n)
                                            hPutStr h "\n"
                                            return h
```

```
  where
    x = xhi − xlo
    y = yhi − ylo
```

We must convert Colour to RGB values for printing. Note that here, n is a direct scale factor and we drop the transparency (not supported by ppm).

```
colourToInts :: Int → Colour → [Int]
colourToInts bits c = [flr r, flr g, flr b]
  where
    n = 2^bits − 1
    flr f = max 0 (min n (floor (realToFrac n ∗ f c')))
    c' = optimizeRgba c

groupRgb :: [Int] → String
groupRgb s = foldr addsp "" s
  where addsp x y = (show x) ++ " " ++ y

showRgb :: Int → Colour → String
showRgb bits = groupRgb ∘ colourToInts bits
```

We can package these into three calls. Two open and close the file. The third is folded over the points to write the data.

```
openPpm :: String → Int → Box → IO(Handle)
openPpm name b bx = do h ← openFile name WriteMode
                       headerPpm h bx n
  where
    n = 2^b −1

closePpm :: Handle → IO ()
closePpm h = do hPutStr h "\n"
                hClose h

writePpm :: Point' p ⇒ Handle → Int → Image' p → Int → p → IO(Int)
writePpm h bits im soFar p = putChunk h soFar $ showRgb bits ∘ im $ p
```

Generating Images

Here we generate the sequence of points, provide utilities for converting between pixels and points, and wrap everything together.

```
dimToBox :: (Int, Int) → Box
dimToBox dim = ((0, 0), dim)

boxToWindow :: Point' p ⇒ p → Box → Window' p
boxToWindow p0 ((xlo, ylo), (xhi, yhi)) =
    (setXy xlo' ylo' p0, setXy xhi' yhi' p0)
  where
    xlo' = fromIntegral xlo
    ylo' = fromIntegral ylo
    xhi' = fromIntegral xhi
    yhi' = fromIntegral yhi

— boxToWindow :: Box → Window Point
— boxToWindow = boxToWindow' origin

dimToWindow :: Point' p ⇒ p → (Int, Int) → Window' p
dimToWindow p0 = boxToWindow p0 ∘ dimToBox

pixelStream :: Point' p ⇒ p → Box → [p]
pixelStream p0 ((xlo, ylo), (xhi, yhi)) =
```

```
  [setXy (0.5 + fromIntegral x) (0.5 + fromIntegral y) p0
    | y ← [yhi−1,yhi−2..ylo], x ← [xlo..xhi−1]]

pixelToPoint :: Point' p ⇒ Window' p → Box → (p → p)
pixelToPoint w b p = mapWindow w (boxToWindow p b) p

{−
ppm :: Window → (Int, Int) → String → Image → IO ()
ppm w dim name im = do h ← openPpm name bits bx
                       foldM (writePpm h bits im') 0 px
                       closePpm h
  where
    bits = 8
    bx = dimToBox dim
    im' = im ∘ pixelToPoint w bx
    px = pixelStream bx
−}
```

The definition of ppm above is not used because we derive it from a more general basis below.

Generalization The following factorization has the benefit of hindsight.

```
type MkBox = (Int, Int) → Box
type MkImage p a = Box → Window' p → Image'' p a → Image'' p a
type PxWrite p p' = Handle → Int → Image' p → [p'] → IO ()
type MkPixels p' = Box → [p']

ppmBase :: MkBox → MkImage p Colour → PxWrite p p' → MkPixels p'
  → Window' p → (Int, Int) → String → Image' p → IO ()
ppmBase mkBox mkImage write mkPixels w dim name im =
  do h ← openPpm name bits bx
     write h bits im' px
     closePpm h
  where
    bits = 8
    full = dimToBox dim
    im' = mkImage full w im
    bx = mkBox dim
    px = mkPixels bx

idBox :: MkBox
idBox = dimToBox

idImage :: Point' p ⇒ MkImage p a
idImage bx w im = im ∘ pixelToPoint w bx

idWrite :: Point' p ⇒ PxWrite p p
idWrite h bits im px = foldM_ (writePpm h bits im) 0 px

idPixels :: Point' p ⇒ p → MkPixels p
idPixels = pixelStream

idPixels' :: MkPixels Point
idPixels' = idPixels origin

ppm :: Window → (Int, Int) → String → Image → IO ()
ppm = ppmBase idBox idImage idWrite idPixels'
```

Transparency Because ppm does not support transparency the opacity will be ignored — this is equivalent to overlaying a transparent image on a black background. If a different background is required then it is trivial to do this using overlay.

```
— opaqueOnWhite = overlay transparentImage white
```

Aliasing Anti–aliasing sub–samples the image. To do this correctly requires information about the range and number of points that will be generated. This means that it must be integrated with the output routines (if the correct parameters are to be automatically carried across).

It's simplest to generate the aliasing points before converting from integers to points within the coordinate system used by the image (ie inbetween pixelStream and pixelToPoint in the output routine above). So the code below generates a list of transforms that work on pixel numbers and then applies the correct transform afterwards.

```
tranAlias :: Point' p ⇒ Int → [p → p]
tranAlias n =
    map mkShift [(x, y) | x ← [1..n], y ← [1..n]]
  where
    mkShift (x, y) = shift (f x) (f y)
    f z = (fromIntegral z − 0.5) / (fromIntegral n) − 0.5
```

This can then be used in an output routine. We reproduce the earlier code, but change the image into a list of parallel pipelines, one for each subsampled point, averaging the result.

It's worth looking at this code in some detail — once it's clear what's happening you'll have understood how to use parallel image pipelines (I hope!) (note that default aliasing is now over 4 samples, rather than 9 as in earlier versions).

```
ppmAlias' ::
  Int → Window → (Int, Int) → String → Image → IO ()
ppmAlias' x = ppmBase idBox mkImage idWrite idPixels'
  where
    mkImage bx w im =
      avg $ map ((im ∘ (pixelToPoint w bx)) ∘) $ tranAlias x

ppmAlias :: Window → (Int, Int) → String → Image → IO ()
ppmAlias = ppmAlias' 2
```

Image Tiling On large images (a 1m square poster, for example, has about 24000 pixels on a side at standard printer resolution) generation of images can take a long time and consume a lot of resources (it can be impossible to view the images with some tools, or memory for Pancito may be limited, or the machine may need to be rebooted reglarly).

These functions allow the image to be split into smaller pieces which, hopefully, can be combined later. Note that (like Box) the indices to tile start at (0,0).

```
tileBox :: (Int, Int) → (Int, Int) → MkBox
tileBox (nx, ny) (ix, iy) (x, y) = ((xlo, ylo), (xhi, yhi))
  where
    xlo = f ix nx x
    xhi = f (ix+1) nx x
```

```
    ylo = f iy ny y
    yhi = f (iy+1) ny y
    f i n mx = if i == n
               then mx
               else i * floor (fromIntegral mx / fromIntegral n)

tile :: (Int, Int) → (Int, Int) →
  Window → (Int, Int) → String → Image → IO ()
tile nxy ixy = ppmBase (tileBox nxy ixy) idImage idWrite idPixels'

tileAlias' :: (Int, Int) → (Int, Int) →
  Int → Window → (Int, Int) → String → Image → IO ()
tileAlias' nxy ixy x = ppmBase (tileBox nxy ixy) mkImage idWrite idPixels'
  where
    mkImage bx w im = avg $ map ((im ∘ (pixelToPoint w bx)) ∘) $ tranAlias x

tileAlias :: (Int, Int) → (Int, Int) →
  Window → (Int, Int) → String → Image → IO ()
tileAlias nxy ixy = tileAlias' nxy ixy 2
```

Progress Meter While generating large images, it would be nice to have some idea of how far we have progressed. Although the following probably makes some unwarranted assumptions about the Haskell implementation, it appears to work well.

```
monitor :: Show a ⇒ Int → [a] → [IO(a)]
monitor n stream = monitor' (0, pc, stream)
  where
    pc = mkPercents n [(0.0, "%: "),
                       (0.00001, "0.001 "),
                       (0.0001, "0.01 "),
                       (0.001, "0.1 "),
                       (0.01, "1 "),
                       (0.02, "2 "),
                       (0.05, "5 "),
                       (0.10, "10 "),
                       (0.20, "20 "),
                       (0.30, "30 "),
                       (0.40, "40 "),
                       (0.50, "50 "),
                       (0.60, "60 "),
                       (0.70, "70 "),
                       (0.80, "80 "),
                       (0.90, "90 "),
                       (0.95, "95 "),
                       (0.98, "98 "),
                       (0.99, "99 "),
                       (0.99999, "100 ")]

monitor' :: Show a ⇒ (Int, [(Int, String)], [a]) → [IO(a)]
monitor' (n, ((n',s):ns), a:as)
 | n ≥ n'   = (unsafeInterleaveIO $ putStr s ≫ hFlush stdout ≫ return a):(monit
 | otherwise = (unsafeInterleaveIO $ return a):(monitor' (n+1,(n',s):ns,as))
monitor' (n, [], a:as) = (unsafeInterleaveIO $ return a):(monitor' (n+1,[],as))
monitor' (_, _, []) = []

mkPercents :: Int → [(Double,String)] → [(Int,String)]
mkPercents n = map (\(x,s) → (floor $ x * n',s))
  where
    n' = fromIntegral n
```

And finally we can lift the previous writePpm and adapt the output functions to include this feedback.

```
monWrite :: Point' p ⇒ PxWrite p (IO p)
monWrite h bits im px =
  do join $ foldM (liftM2 (writePpm h bits im)) (return 0) px
     putStr "done\n"

monPixels :: Point' p ⇒ p → MkPixels (IO p)
monPixels p0 bx = monitor (countBox bx) $ pixelStream p0 bx

monPixels' :: MkPixels (IO Point)
monPixels' = monPixels origin

countBox :: Box → Int
countBox ((xlo, ylo), (xhi, yhi)) = (xhi−xlo)∗(yhi−ylo)

ppm_ :: Window → (Int, Int) → String → Image → IO ()
ppm_ = ppmBase idBox idImage monWrite monPixels'
```

Hopefully the apparently arbitrary abstraction of ppmBase is now explained.

```
ppmAlias'_ ::
  Int → Window → (Int, Int) → String → Image→ IO ()
ppmAlias'_ x = ppmBase idBox mkImage monWrite monPixels'
  where
    mkImage bx w im = avg $ map ((im ∘ (pixelToPoint w bx)) ∘) $ tranAlias x

ppmAlias_ :: Window → (Int, Int) → String → Image → IO ()
ppmAlias_ = ppmAlias'_ 2

tile_ :: (Int, Int) → (Int, Int) →
  Window → (Int, Int) → String → Image → IO ()
tile_ nxy ixy = ppmBase (tileBox nxy ixy) idImage monWrite monPixels'

tileAlias'_ :: (Int, Int) → (Int, Int) →
  Int → Window → (Int, Int) → String → Image → IO ()
tileAlias'_ nxy ixy x =
    ppmBase (tileBox nxy ixy) mkImage monWrite monPixels'
  where
    mkImage bx w im = avg $ map ((im ∘ (pixelToPoint w bx)) ∘) $ tranAlias x

tileAlias_ :: (Int, Int) → (Int, Int) →
  Window → (Int, Int) → String → Image → IO ()
tileAlias_ nxy ixy = tileAlias'_ nxy ixy 2
```

Reprocess

Header This module allows you to read images back into the system (and reprocess them). Only ASCII ppm images can be read — check out pnmtoplainpnm from the netpbm package (this is the same format that Pancito uses for output).

```
module Reprocess (
  readPpm, wrapArray, toDim, readPpm_ {− , subPpm, subPpm_ −},
  monitor
) where
import Point
import Colour
```

```
import Pancito2
import Word
import IO
import Array
import System.IO.Unsafe
```

Arrays The image is read into an array. Functions here allow that array to be used as an Image. This function might be useful in its own right one day.

TODO - wraparray unification? (later - what was i thinking here?)

```
wrapArray :: Point' p ⇒ Window' p → a → Array (Int, Int) a → Image'' p a
wrapArray box c a p = if (winRegion box) p
                        then a!(quant p box (bounds a))
                        else c

quant :: Point' p ⇒ p → Window' p → ((Int, Int), (Int, Int)) → (Int, Int)
quant p (lo,hi) ((ilo,jlo),(ihi,jhi)) = (i, j)
  where
    x' = x p
    y' = y p
    xlo = x lo
    ylo = y lo
    xhi = x hi
    yhi = y hi
    dx = (x' − xlo) / (xhi − xlo)
    dy = (y' − ylo) / (yhi − ylo)
    ni = ihi − ilo + 1
    nj = jhi − jlo + 1
    i = min (ilo + floor(dx ∗ fromIntegral ni)) ihi
    j = min (jlo + floor(dy ∗ fromIntegral nj)) jhi

wrapArray8 :: Point' p ⇒
  Window' p → Colour → Array (Int, Int, Int) Word8 → Image' p
wrapArray8 box c a p = if (winRegion box) p
                         then word8ToColour a (quant p box ((1, 1), (nx, ny)))
                         else c
  where
    ((1, 1, 1), (nx, ny, 3)) = bounds a

word8ToColour :: Array (Int, Int, Int) Word8 → (Int, Int) → Colour
word8ToColour a (x, y) = rgba r g b 1.0
  where
    r = fromIntegral (a!(x, y, 1)) / 255.0
    g = fromIntegral (a!(x, y, 2)) / 255.0
    b = fromIntegral (a!(x, y, 3)) / 255.0
```

Parsing the File Simple utilities to parse the file contents.

```
header :: String → ((Int, Int), Int, [String])
header = headerParams ∘ checkType ∘ words ∘ dropComment

headerParams (nx:ny:n:s) = ((read nx, read ny), read n, s)
headerParams _           = error "malformed header"

checkType ("P3":s) = s
checkType _        = error "incorrect file format"

dropComment []      = []
```

```
dropComment ('#':s) = dropComment' s
dropComment (c:s)   = c:dropComment s

dropComment' []         = []
dropComment' (c@'\n':s) = c:dropComment s
dropComment' (c:s)      = dropComment' s
```

Read the Data Put everything together.

```
type PxFilter' a = [((Int, Int), a)] → [((Int, Int), a)]
type PxFilter a = (Int, Int) → ((Int, Int), PxFilter' a)

eagerParse8 :: PxFilter (String, String, String) → Handle
  → IO (Array (Int, Int, Int) Word8, (Int, Int))
eagerParse8 f h = do str ← hGetContents h
                     (dim, mx, toks) ← return $ header str
                     (dim', f') ← return $ f dim
                     return $! (pixels8 dim dim' f' toks, dim')

zipAll (a:as) (b:bs) = (a, b):(zipAll as bs)
zipAll []     []     = []
zipAll _      _      = error "missing data"

group3 (a:b:c:s) = (a, b, c):(group3 s)
group3 []        = []
group3 _         = error "malformed data"

pixels8 :: (Int, Int) → (Int, Int) → PxFilter' (String, String, String)
  → [String] → Array (Int, Int, Int) Word8
pixels8 (nx, ny) (dx, dy) filter s =
    array bnds ∘ parse8 ∘ filter $ zipAll xy pix
  where
    bnds = ((1, 1, 1), (dx, dy, 3))
    xy = map (\(y,x) → (x, ny–y+1)) (range ((1,1), (ny,nx)))
    pix = group3 s

parse8 :: [((Int, Int), (String, String, String))]
  → [((Int, Int, Int), Word8)]
parse8 []                      = []
parse8 (((x, y), (r, g, b)):ps) = ((x, y, 1), read r):
                                  ((x, y, 2), read g):
                                  ((x, y, 3), read b):(parse8 ps)

readPpm :: Point' p ⇒
  String → Window' p → Colour → IO (Image' p, (Int, Int))
readPpm name w bg = do h ← openFile name ReadMode
                       (a, dim) ← eagerParse8 idFilter h
                       return (wrapArray8 w bg a, dim)
  where
    idFilter dim = (dim, id)
```

We can modify the pixel stream to generate a progress bar, just as with the output routines.

TODO - merge with OP code

```
monitorF :: PxFilter a
monitorF dim@(nx, ny) = (dim, monitor (nx * ny))

monitor :: Int → [a] → [a]
monitor n stream = monitor' (0, pc, stream)
```

```
  where
    pc = mkPercents n [(0.0, "%: "),
                       (0.00001, "0.001 "),
                       (0.0001, "0.01 "),
                       (0.001, "0.1 "),
                       (0.01, "1 "),
                       (0.02, "2 "),
                       (0.05, "5 "),
                       (0.10, "10 "),
                       (0.20, "20 "),
                       (0.30, "30 "),
                       (0.40, "40 "),
                       (0.50, "50 "),
                       (0.60, "60 "),
                       (0.70, "70 "),
                       (0.80, "80 "),
                       (0.90, "90 "),
                       (0.95, "95 "),
                       (0.98, "98 "),
                       (0.99, "99 "),
                       (0.99999, "100 ")]

monitor' :: (Int, [(Int, String)], [a]) → [a]
monitor' (n, pc@((n', s):ns), a:as)
 | n ≥ n'             = (unsafePerformIO $ do putStr s
                                              hFlush stdout
                                              return a):
                          (monitor' (n+1, ns, as))
 | otherwise          = a:(monitor' (n+1, pc, as))
monitor' (n, [], a:as) = a:(monitor' (n+1, [], as))
monitor' (_, _, [])    = (unsafePerformIO $ do putStrLn "done"
                                              return [])

mkPercents :: Int → [(Double,String)] → [(Int,String)]
mkPercents n = map (\(x,s) → (floor $ x * n',s))
  where
    n' = fromIntegral n

readPpm_ :: Point' p ⇒
  String → Window' p → Colour → IO (Image' p, (Int, Int))
readPpm_ name w bg = do h ← openFile name ReadMode
                        (a, dim) ← eagerParse8 monitorF h
                        return (wrapArray8 w bg a, dim)
```

And we can subsample the image with a suitable filter.

```
everyN :: Int → PxFilter a
everyN n (nx, ny) = ((nx `div` n, ny `div` n), everyN' n)

everyN' n []                 = []
everyN' n (((x, y), pix):as) = if ok
                               then ((x', y'), pix):(everyN' n as)
                               else everyN' n as
  where
    (x', y') = (x `div` n, y `div` n)
    ok = n * x' == x && n * y' == y

stackF :: PxFilter a → PxFilter a → PxFilter a
stackF f' f dim = (dim'', f''' ∘ f'')
  where
    (dim', f'') = f dim
```

```
    (dim'', f''') = f' dim'

{-
TOD merge

subSample :: Int → (PxFilter, DimFilter)
subSample n = (subPix n, subDim n)

subPix :: Int → PxFilter
subPix n ((x', y'), c) = if mtch then Just ((x'', y''), c) else Nothing
  where
    x'' = x' `div` n
    y'' = y' `div` n
    mtch = unsafePerformIO $ do m ← return $ x' == x'' * n && y' == y'' * n
                                if m then print (x',y',x'',y'') else return ()
                                return m

subDim :: Int → DimFilter
subDim n (nx, ny) = (nx `div` n, ny `div` n)

subPpm :: Int → String → Window' → Colour → IO (Image, (Int, Int))
subPpm n = ppmBase' idRead idPixels (subSample n)

subPpm_ :: Int → String → Window' → Colour → IO (Image, (Int, Int))
subPpm_ n = ppmBase' monRead monPixels (subSample n)
-}
```

We might want to have the image as a function of its pixel coordinates.

```
toDim :: Point' p ⇒ p → (Int, Int) → Window' p → Transform' p
toDim p0 = flip mapWindow ∘ dimToWindow p0
```

Example This is fairly obvious, I hope.

TODO fix

```
{-
module Main where
main :: IO ()
main = do img ← readPpm "test-in.ppm" square01 white
          ppm square11 (20, 20) "test-out.ppm" img
-}
```

Flux

Header This module supports images and transformations which conserve "flux" in a very general sense. More accurately, they work with (Point, Scaling) tuples, where Scaling is a factor that represents the cumulative Jacobean of the transformations made.

What is a Jacobean? In simple terms, it's the ratio of pixel areas before and after a transform. It's useful when modelling some physical property that is fixed per unit area (in the case of images, typically light). So , for example, if you magnify something you're spreading the photons that were coming towards your eye over a wider area, and the image should appear darker.

```
module Flux (
  FPoint (FPoint), p, z, setP, setZ, fOrigin, fppm, fppm_,
```

```
  point, line,
  Render, idRender, mkRender,
  renderSat, fSat, renderHue, fHue, renderVal, fVal, renderVal', fVal',
  fTint
) where

import Colour
import Point
import Pancito2


data FPoint = FPoint Point Double

setP :: Point → FPoint → FPoint
setP p (FPoint _ z) = FPoint p z

setZ :: Double → FPoint → FPoint
setZ z (FPoint p _) = FPoint p z

p :: FPoint → Point
p (FPoint p' _) = p'

z :: FPoint → Double
z (FPoint _ z') = z'

instance Point' FPoint where
  x = x ∘ p
  y = y ∘ p
  d = d ∘ p
  t = t ∘ p
  optimizeCartesian (FPoint p z) = FPoint (optimizeCartesian p) z
  optimizePolar (FPoint p z) = FPoint (optimizePolar p) z
  setXy x y (FPoint p z) = FPoint (setXy x y p) z
  setDt d t (FPoint p z) = FPoint (setDt d t p) z

instance Eq FPoint where
  p1 == p2 = (p p1) == (p p2)

instance Show FPoint where
  show (FPoint p z) =
    "[" ++ (show $ x p) ++ "," ++ (show $ y p) ++ "," ++ (show z) ++ "]"

fOrigin :: FPoint
fOrigin = FPoint origin 1.0
```

Output It's easy to build output routines from the base code in the Pancito2 module.

```
fppm :: Window' FPoint → (Int, Int) → String → Image' FPoint → IO ()
fppm = ppmBase idBox idImage idWrite (idPixels fOrigin)

fppm_ :: Window' FPoint → (Int, Int) → String → Image' FPoint → IO ()
fppm_ = ppmBase idBox idImage monWrite (monPixels fOrigin)
```

Lensing The basic transforms for a lens. The first parameter gives the strength of the lensing, the second gives the scale.

```
f  a b x = x * (1.0 + (a * x) / (x**2 + b**2))
f' a b x = 1.0
```

```
          + 2.0 * a * x / (x**2 + b**2)
          - 2.0 * a * x**3 / (x**2 + b**2)**2
```

A point lens.

```
point :: (Double, Double) → FPoint → Transform' FPoint
point (a, b) c fp = setZ z' (c `add'` delta')
  where
    delta = fp `sub'` c
    dist = d delta
    dist' = f a b dist
    delta' = setD dist' delta
    z' = (z fp) * f' a b dist
```

A linear lens.

```
line :: (Double, Double) → Line FPoint → Transform' FPoint
line (a, b) ln fp = setZ z' (near `add'` delta')
  where
    near = nearest ln fp
    delta = fp `sub'` near
    dist = d delta
    dist' = f a b dist
    delta' = setD dist' delta
    z' = (z fp) * f' a b dist
```

Rendering The FPoint based transforms construct a filter that is applied to an underlying Image. The conversion from Image to Image' FPoint involves interpreting the Jacobean.

```
type Render = Image → Image' FPoint

idRender :: Render
idRender im fp = im (p fp)

mkRender :: VTint' FPoint → Render
mkRender vt im fp = vt fp $ idRender im fp

renderSat :: Render
renderSat = mkRender fSat

fSat :: VTint' FPoint
fSat fp = saturate (z fp - 1.0)

renderVal :: Render
renderVal = mkRender fVal

fVal :: VTint' FPoint
fVal fp = brighten (z fp - 1.0)

renderVal' :: Render
renderVal' = mkRender fVal'

fVal' :: VTint' FPoint
fVal' fp = brighten' (z fp)

renderHue :: Render
renderHue = mkRender fHue

fHue :: VTint' FPoint
fHue fp = rotateHue (z fp - 1.0)
```

Conversion Rendering converts an image, but we also need to convert other types.

```
fTint :: VTint → VTint' FPoint
fTint vt fp = vt (p fp)
```

Pixels

Header These routines support images that display an image using "TV pixels".

```
module Pixels (
  toPix, fromPix, fuzz
) where

import Common
import Point
import Colour
import Pancito2
import Utilities
```

Coordinate conversion Conversion to and from the frame of the image that is used as a map of pixels (ie the image "on the TV").

```
toPix, fromPix :: (Int, Int) → Window → Transform
toPix dim w = mapWindow w (dimToWindow dim)
fromPix dim w = mapWindow (dimToWindow dim) w
```

RGB Fuzz takes two random images (see Utilities) and uses them to generate a filter that maps each pixel in the underlying image into a "sub-pixel" - the phosphor dots.

There are two different levels of pixelisation here. First, the pixelated image (possibly text). Second, the pixels on the display device (phosphor dots), which sub-sample the image pixels. The adegree of sub-sampling is defined by the first argument to fuzz.

There's some bleeding of information between phosphor dots. I'm not sure it's realistic, but it looks OK.

```
fuzz :: (Int, Int) → Image' Double → Image' Double → Image → Image
fuzz (nx, ny) rn1 rn2 im p =  clr
  where
    (nx', ny') = (fromIntegral nx, fromIntegral ny)
    p' = scale nx' ny' p
    im' = im ∘ unscale nx' ny'
    c1 = centre p'
    of1 = sub' p' c1
    ang = t of1
    c2 = centre $ add' c1 $ polar 1.49 ang
    of2 = sub' p' c2
    cl1 = im' c1
    cl2 = im' c2
    r1 = max (abs $ x of1) (abs $ y of1)
    r2 = max (abs $ x of2) (abs $ y of2)
    z1 = 0.3 + 0.5 * rn1 p
    z2 = 0.1 * rn2 p
    x3 = roll 0.0 3.0 (x p')
    pxl cl = if x3 < 1.0
                then rgba (r cl) 0.0 0.0 1.0
```

```
                else if x3 < 2.0 then rgba 0.0 (g cl) 0.0 1.0
                                 else rgba 0.0 0.0 (b cl) 1.0
    clr = if v cl1 > 0.01 then shd r1 (pxl cl1) else shd r2 (pxl cl2)
    shd r c = if z1 > r && v c > 0.01
                then c
                else rgba 0.05 0.05 0.05 1.0
```

Test This section describes a "typical" image.

Modules As well as the Pancito modules, we use random numbers.

```
module Main where

import Random
import Pancito2
import Colour
import Point
```

Random Squares This code is a bit clunky — at one point it had an extra predicate in the comprehension that generated the list of colours (the idea was to test for suitable colour combinations). Without that test the number of random numbers required is fixed and the code could probably be simplified (ie there's no backtracking).

```
mkImages :: Int → Int → [Double] → [Bool] → [Image]
mkImages nx ny ran flip =
  map mkSquare $ zip [(x, y) | x ← [(−1)∗nx..nx], y ← [(−1)∗ny..ny]]
                     (map mkBase $ zip [c | c ← threeColours ran]
                                       (double flip))

double :: [a] → [(a,a)]
double ran = (r1, r2) : double ran'
  where r1:r2:ran' = ran

triple :: [a] → [(a,a,a)]
triple ran = (r1, r2, r3) : triple ran'
  where r1:r2:r3:ran' = ran

mkColour :: (Double, Double, Double) → Colour
mkColour (r1, r2, r3) = hsva (2 ∗ pi ∗ r1) r2 r3 1

threeColours :: [Double] → [(Colour, Colour, Colour)]
threeColours ran = triple $ map mkColour $ triple ran
```

This is the function that varies the colour across a square. There is an implicit asumption that the colours are in HSVA format (addition in RGB would look different). This is a gotcha I discuss earlier. I could use optimizeHsva, but since I know that mkColour is generating HSVA values, there's no real need.

```
mkBase :: ((Colour, Colour, Colour), (Bool, Bool)) → Image
mkBase ((c1, c2, c3), (b1, b2)) p =
    (cmb b2 (cmb b1 c1 c2 (x p)) c3 (y p))
  where
    cmb b c1 c2 z = (if b then add else sub) c1 (cMap ((z ∗ 0.2) ∗) c2)

mkSquare :: ((Int, Int), Image) → Image
mkSquare ((x', y'), im) = (square im) ∘ shift x'' y''
  where
```

```
    x'' = 2 * fromIntegral x'
    y'' = 2 * fromIntegral y'

box :: Region
box p = abs (x p) < 0.7 && abs (y p) < 0.7

square :: Image → Image
square im p = if contains box p
                then im p
                else transparent
```

The arcPixel function quantizes the image in small arcs, giving the impression that the screen is made from pixels in a circular pattern. At least, that was the original intentions (and it does work on images which change colour fairly rapidly). Here, however, it simply crinkles the edges of the gem-like squares.

```
arcPixel :: Transform
arcPixel p = setDt qd qt p
  where
    d' = d p
    t' = t p
    qd = quant n d'
    qt = if qd < tiny then 0 else (quant n (qd * (t' + z))) / qd - z
    n = 40
    z = 0.1

quant :: Double → Double → Double
quant n x = (fromIntegral (round (n * x))) / n
```

Finally, combine everything.

```
im :: Image
im = (over black $ mkImages n n ran flip) ∘
       expand (2 * (1 + fromIntegral n)) ∘ arcPixel
  where
    n = 2
    ran = randoms $ mkStdGen 1
    flip = randoms $ mkStdGen 2

main :: IO ()
main = ppmAlias_ square11 (200, 200) "test.ppm" im
```

To run this code, type (exact details depend on your compiler etc.):

```
ghc --make Test -O -o Test
./Test
```

To generate the image (200x200 pixels, anti-aliased) takes less than a minute on a 2GHz desktop. Without anti-aliasing it should take just over 1/4 the time.

I use xloadimage (or xli) and Gimp or NetPBM to view and manipulate the result. Intermediate results can be displayed with xloadimage (which will display partial images).

Utilities

Header This module is, strictly, not part of Pancito because it changes "at random" (ie within releases). It contains functions that I find useful. Often these are pulled out of previous image scripts; sometimes existing functions are made more general.

```
module Utilities (
  wedge, checks, centre,
  ranList, ranList',
  logo, logo', logo'', shiftLogo,
  Position (BottomRight, BottomLeft, TopRight, TopLeft),
  ranImage, parallel, wrap, ranImage',
  border,
  nLevels, mkQuant, nBright,
  limits, rescale, feather, gamma'
) where

import Common
import Point
import Colour
import Pancito2
import Random
import Reprocess
import Array
```

Motifs One process that interests me is tiling - repeating a motif across an image. I do this by defining a function that generates a motif within square11 and then using pixelate11 (both functions in the Points module).

Wedge can be used to produce "striped" images where, for example, stripe width depends on intensity.

```
wedge :: (Double, Double) → (Colour, Colour) → Image
wedge (lo, hi) (fg, bg) p = if inside then fg else bg
 where
   x' = abs (x p)
   y' = y p
   x'' = lo + (hi − lo) ∗ (y' + 1.0) / 2.0
   inside = x' < x''
```

Other Tiling Functions "Round" a point to 0.5,1.5,...

```
centre :: Point' p ⇒ p → p
centre p = setXy x' y' p
  where
    x' = 0.5 + fromIntegral (floor (x p))
    y' = 0.5 + fromIntegral (floor (y p))
```

Checkerboard Useful for backgrounds, to give scaling, while working on images.

```
checks :: Double → Colour → Colour → Image
checks sd c1 c2 p = if even (ix+iy) then c1 else c2
  where
    ix = index (x p)
    iy = index (y p)
    index z = floor (z / sd)
```

Logo This assumes that logo.ppm exists in the current directory. The shiftLogo function can be used to place the logo, scaled to a given width, within the main image's coordinates.

```
logo :: IO (Image, (Int, Int))
logo = do (img, dim) ← readPpm "logo.ppm" square01 transparent
          return (img ∘ toDim origin dim square01, dim)

data Position = BottomRight Point | BottomLeft Point
              | TopRight Point | TopLeft Point

shiftLogo :: Double → Position → (Int, Int) → Transform
shiftLogo wd pos dim@(nx,ny) =
    mapWindow (dimToWindow origin dim)
      (case pos of
        (BottomRight p) → (cartesian (x p − wd) (y p),
                           cartesian (x p) (y p + ht))
        (BottomLeft p) → (cartesian (x p + wd) (y p + ht),
                          cartesian (x p) (y p))
        (TopRight p) → (cartesian (x p) (y p),
                        cartesian (x p − wd) (y p + ht))
        (TopLeft p) → (cartesian (x p + wd) (y p),
                       cartesian (x p) (y p − ht)))
  where
    ht = wd ∗ fromIntegral ny / fromIntegral nx

logo' :: Tint → Tint → Tint → ((Int, Int) → Transform) → IO VTint
logo' fg bg cs tr = do (lg, dim) ← logo
                       return $ tnt dim lg tr
  where
    tnt dim lg tr p = if winRegion (dimToWindow origin dim) p'
                        then ltint fg bg cs $ lg p'
                        else id
      where
        w = dimToWindow origin
        p' = tr dim p

logo'' :: ((Int, Int) → Transform) → IO VTint
logo'' tr = logo' (lighten 0.2) (darken 0.2) (overlay orange) tr
  where
    orange = rgba (214.0/255.0) (53.0/255.0) (12.0/255.0) 0.2

ltint :: Tint → Tint → Tint → Colour → Tint
ltint fg bg cs c = if r' < 0.1
                     then bg
                     else if r' > 0.9
                          then fg
                          else cs
  where
    r' = r c
```

Random Numbers An infinite list of random generators is a useful thing to fold over when generating lists of random processes.

```
ranList :: StdGen → [StdGen]
ranList g = g' : ranList g''
  where
    (g', g'') = split g

ranList' :: Int → [StdGen]
ranList' n = ranList (mkStdGen n)
```

Sometimes it's useful to have a random number for each pixel. We can generalise this to an image with any type of contents.

```
ranImage :: StdGen → (Int, Int) → Window → Image'' Point Double
ranImage ran dim@(nx, ny) w = wrapArray w 0.0 a
  where
    bnds = ((1,1), dim)
    asc = zip (range bnds)
              (randomRs (0.0, 1.0) ran)
    a = array bnds $ monitor (nx * ny) asc

parallel :: Image'' p a → Image'' p b → Image'' p (a, b)
parallel im1 im2 p = (im1 p, im2 p)
```

Random images are expensive to generate. A smaller image plus wrapping of coordinates may be all you need. The function ranImage' takes care of worrying that the pixel density is the same as the image you are working on.

```
wrap :: Point' p ⇒ Window' p → Transform' p
wrap (lo, hi) p = p'
  where
    (xlo, ylo) = (x lo, y lo)
    (xhi, yhi) = (x hi, y hi)
    p' = setXy (roll xlo xhi $ x p) (roll ylo yhi $ y p) p

ranImage' ::
  StdGen → (Int, Int) → (Int, Int) → Window → Image'' Point Double
ranImage' ran full dim w = ranImage ran dim w ∘ wrap w ∘ zm
  where
    zm = mapWindow (dimToWindow origin full) (dimToWindow origin dim)
```

Dark Border This is used to border the X-Ray image. The idea is to have a border that fades to black with some structure, reflecting the original image. The hard-coded power of 32 defines the width in a coordinate-dependent form (this should be seeparated into a parameter if this is re-used).

```
wgtSum :: Point' p ⇒ Window' p → Image' p → p → Double
wgtSum (lo, hi) im p = wgtSum' dx (setXy xlo y' p) im p +
                       wgtSum' dx (setXy xhi y' p) im p +
                       wgtSum' dy (setXy x' ylo p) im p +
                       wgtSum' dy (setXy x' yhi p) im p
  where
    (xlo, ylo) = xy lo
    (xhi, yhi) = xy hi
    (x', y') = xy p
    (dx, dy) = (xhi − xlo, yhi − ylo)

wgtSum' :: Point' p ⇒ Double → p → Image' p → p → Double
wgtSum' nrm p1 im p2 =
  (1.0 − v (im p1)) * ((nrm − abs (d (sub' p1 p2))) / nrm)^32

border :: Point' p ⇒ Window' p → Filter' p
border w im p = darken (x1+x2) $ im p
  where
    x1 = wgtSum w im p
    x2 = wgtSum w (flat black) p
```

Quantisation Helps generate contours.

```
nLevels :: (Double, Double) → Int → Double → Double
nLevels (lo, hi) n x = x'
  where
    range = hi − lo
    bin = range / (fromIntegral n)
    norm = (fromIntegral n) ∗ (x − lo) / range
    quant = fromIntegral $ floor norm
    quant' = min (fromIntegral $ n−1) $ max 0.0 quant
    x' = 0.5 ∗ bin + lo + quant' ∗ bin

mkQuant :: (Colour → Double) → (Double → Colour → Colour)
  → (Double → Double) → Tint
mkQuant get set quant c = set (quant $ get c) c

nBright :: Int → Tint
nBright n = mkQuant b setB (nLevels (0.0, 1.0) n)
```

Range Measure the range of some value in an image.

```
limits :: Point' p ⇒
  Window' p → (Int, Int) → (Double, Double) → (a → Double) → p
  → Image'' p a → (Double, Double)
limits w dim mnmx f p0 im = measure mnmx f im' px
  where
    full = dimToBox dim
    im' = idImage full w im
    bx = idBox dim
    px = idPixels p0 bx

measure :: Point' p ⇒
  (Double, Double) → (a → Double) → Image'' p a → [p] → (Double, Double)
measure mnmx f im ps = foldl getLim mnmx ps
  where
    getLim (mn, mx) p = (min mn a, max mx a)
      where
        a = f $ im p
```

Rescale Transform one range to another.

```
rescale :: (Double, Double) → (Double, Double) → Double → Double
rescale (inlo, inhi) (outlo, outhi) x =
  outlo + (outhi − outlo) ∗ ((x − inlo) / (inhi − inlo))
```

Feather Fix the alpha channel to match the content (to give progressive overlaying where black).

```
feather :: Tint
feather c = setA (max 0.0 (max (r c) (max (g c) (b c)))) c
```

Gamma Apply a generalised gamma correction.

```
gamma' :: (a → Double) → (Double → a → a) → Double → Tint' a a
gamma' get set g a = set ((get a)**g) a
```

Common Some very basic common functions.

```
module Common (
  clip, clip0, clip01, roll, roll2pi
) where
```

```
roll :: Double → Double → Double → Double
roll lo hi x | x ≥ hi   = roll lo hi (x − d)
             | x < lo   = roll lo hi (x + d)
             | otherwise = x
  where
    d = hi − lo

roll2pi :: Double → Double
roll2pi = roll 0.0 (2.0 * pi)

clip :: Double → Double → Double → Double
clip lo hi x | x > hi    = hi
             | x < lo    = lo
             | otherwise = x

clip0 :: Double → Double → Double
clip0 = clip 0

clip01 :: Double → Double
clip01 = clip0 1
```

Documentation

You can generate a postscript document from the Pancito2 files by typing:

```
latex Pancito2.tex
dvips Pancito2 -o Pancito2.ps
```

This expects to find the haskell.sty file, which you can get from my web site if you do not already have it. That, in turn, expects to find various style files that are installed by default if you use the tetex package (I use the version included in the "testing" Debian distribution; older versions may not work correctly).

Pancito2.tex also requires the source files Pancito2.lhs, Point.lhs, Colour.lhs and Test.lhs.

Credits

Thanks to Conal Elliot for Pan and all the people on c.l.functional and the Haskell mailing list for replying to questions.

Licencing Conditions