

Pancito

Introduction

Pan and Pancito Pancito was inspired by the Pan project¹. Like Pan it provides support for manipulating *functional images* in Haskell. However, there are many differences:

- Pancito does not support an interactive viewer or animations. Instead, images are written directly to disk as ppm files.
- Pancito is pure Haskell. Pan is a language written inside Haskell that is translated to C and then compiled for speed.
- Pancito does not have anything like as many predefined functions as Pan.

Manifesto Given all that, you might wonder why you should use Pancito rather than Pan. I think there are three reasons. First, it runs anywhere Haskell runs (Pan only runs on Windows machines with a Microsoft C compiler). Second, I think it's easier to understand. There's no confusing intermediate language, for example. Third, the more spartan functionality forces you to create. Good art is helped by constraints. Too much computer art (and too many of the Pan examples) are impressive because they look technically difficult – that's not what art is about. If you feel otherwise, you might want to save some effort and move to Pan now!

Functional Images What does Pancito take from Pan? Both packages use Haskell. Both support the idea that code that creates elegant images should be elegant code. And finally, both use functional images.

An image in Pan or Pancito is a function. Given a position it specifies a colour. To display (ie. write to a file) an image it is necessary for something to iterate over the different pixel positions, requesting the colour at each point. This is what Pancito does, writing a file to disk. You can then view or manipulate this file with another program (eg. `xloadimage` or `netpbm`).

Actually, Pancito does more than that. It also provides a basic framework for writing these images/functions. This includes defining how colours are represented and providing some ways of manipulating them (+, for example, will overlay two colours).

Functional images are interesting because they make some things that are difficult with traditional packages very easy. On the other hand they makes other things (eg. drawing lines) much harder. A whole new bunch of constraints and freedoms for computer art — that has to be good news.

¹<http://research.microsoft.com/~conal/Pan/>

Haskell Haskell is an elegant functional language. If you haven't used a functional language before, Pancito might be a good way to learn. For more general information, point your browser at www.haskell.org.

This document, which describes the Pancito module, includes all the source code. Haskell compilers (at least, hugs, ghc and nhc98) can compile Pancito directly from the document you are reading. Keeping the code and documentation together makes everything easier to maintain and, hopefully, easier to understand. Unfortunately, it also means that even the boring, ugly bits of code, like this module definition, must be present.

```
module Pancito (
    Image, Colour, Clr, Point, Filter, Transform,
    red, green, blue, black, white,
    cyan, magenta, yellow, grey,
    interp, lighten, darken,
    solid, transparent, opacity,
    underlay, peel, normalize,
    Window, mapWindow, contains,
    origin, unit, nUnit, square01, square11,
    shift, scale,
    colourStream, directSample, colourToInts, groupRGB,
    writePpmBase, writePpm,
    foldC, zipWithC, zipC, boolC, flipC,
    quantisedList, addImage, colourImage, nullImage, sumComponents,
    doEx1, doEx2, doEx3,
    windowedImage,
    Polar, distSq, dist, angle, fromPolar, toPolar,
    similar, antiAlias, writePpmAlias
) where

import IO
import Monad
```

Basic Types

Image OK, useful code starts here. As I've already said, Pancito is distinctive because it considers images to be functions. Give an image a point and it gives you a colour:

```
type Image = Point → Colour
```

Filter, Point, Transform Filters transform images. Note that this transformation is completely general; the image returned by a filter may manipulate the point before passing it to the "enclosed" image (allowing scaling, for example), as well as manipulating the resulting colour. This implies that a point, despite being similar to a pixel's coordinate, is identified by a pair of real (ie. floating point, Double) numbers rather than integers. Since manipulating coordinates alone is common, we also define Transform.

```
type Filter = Image → Image
type Point = (Double, Double)
type Transform = Point → Point
```

Colour We must define how we represent colours. The four values used by the RGBA constructor here represent red, green, blue and alpha (opacity) (in that order). In normal use, each should lie between 0 and 1. Unlike Pan, alpha is *not* pre-multiplied.

```
data Clr a = RGBA a a a a
type Colour = Clr Double
```

Colours

Values We can define an initial set of solid colours:

```
solid :: Double → Double → Double → Colour
solid r g b = RGBA r g b 1

black, red, green, blue, white :: Colour
black = solid 0 0 0
red = solid 1 0 0
green = solid 0 1 0
blue = solid 0 0 1
white = solid 1 1 1
```

Interpolation Colours can be interpolated (this preserves alpha from the *first* colour, allowing functions like lighten and darken that leave transparency unchanged):

```
interp :: Double → Colour → Colour → Colour
interp frac (RGBA r1 g1 b1 a) (RGBA r2 g2 b2 _) =
  RGBA (intp r1 r2) (intp g1 g2) (intp b1 b2) a
  where
    intp x y = x + frac * (y - x)

lighten, darken :: Double → Colour → Colour
lighten frac c = interp frac c white
darken frac c = interp frac c black

cyan, magenta, yellow, grey :: Colour
cyan = interp 0.5 blue green
magenta = interp 0.5 red blue
yellow = interp 0.6 green red
grey = interp 0.5 black white
```

Transparency A colour with an alpha value less than 1 is at least partially transparent.

```
opacity :: Double → Colour → Colour
opacity a (RGBA r g b _) = RGBA r g b a

transparent :: Colour
transparent = opacity 0 black
```

To overlay one colour on top of another, use the following function (or `+`, which is defined as the same thing later). The name reflects the order of the arguments (background, foreground) which is chosen for sensible folding (repeated application of a function to a list; a common process in Haskell programs). The inverse operation, removing a colour to reveal what is underneath, is called `peel`. Since this can produce negative

colours, normalize guarantees that result is within the expected ranges (- is overloaded as normalize . peel). Beware — peeling a solid color will (not surprisingly) provoke an error.

```

underlay :: Fractional a => Clr a -> Clr a -> Clr a
underlay (RGBA bgR bgG bgB bgA) (RGBA fgR fgG fgB fgA) =
  (RGBA (intp bgR fgR) (intp bgG fgG) (intp bgB fgB) a)
  where
    intp bg fg = fgA * fg + (1 - fgA) * bg
    a = bgA + (1 - bgA) * fgA

peel :: Fractional a => Clr a -> Clr a -> Clr a
peel (RGBA r g b a) (RGBA fgR fgG fgB fgA) =
  (RGBA (pl r fgR) (pl g fgG) (pl b fgB) (pl a 1))
  where pl both fg = (both - fg * fgA) / (1 - fgA)

normalize :: (Num a, Ord a) => Clr a -> Clr a
normalize c = fmap norm c
  where norm x = min 1 (max 0 x)

```

Transforms Be careful here. It's easy to get confused about transforms. I've found that the best way to think about them is to:

- Be very clear about the range of coordinates an image expects. I find that it's easiest to work with images that extend from “bottom left” at (0,0) or (-1,-1) to “top right” at (1,1).

```
type Window = (Point, Point)
```

```

origin, unit, nUnit :: Point
origin = (0.0, 0.0)
unit = (1.0, 1.0)
nUnit = (-1.0, -1.0)

```

```

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

```

- Picture the process of image generation as passing coordinates into a series of transformations “from the right”. For code like

```
image . transform1 . transform2
```

for example, the initial coordinates (which cover the range specified to the output routines) are passed through transform2, then transform1, and finally arrive at image (which calculates the colour at this transformed coordinate).

That may all seem obvious, so, at the risk of making things worse, it's worth pointing out that the danger comes when you confuse scaling the coordinates with scaling the image (making coordinates bigger makes the displayed image smaller) or confuse shifting the coordinates with shifting the image (moving the coordinates further right moves the displayed image left).

If you remember that we transform *coordinates*, not images, then the functions below make sense.

```
shift :: (Double, Double) → Transform
shift (dx, dy) (x, y) = (x + dx, y + dy)
```

```
scale :: (Double, Double) → Transform
scale (fx, fy) (x, y) = (x * fx, y * fy)
```

You might want to look at the Pan documentation for more ambitious transformations as this is one area where functional images are very flexible.

A useful way of representing the transforms above is to specify two rectangles — one in the image and one in the display — that the coordinates should map between.

```
mapWindow :: Window → Window → Transform
mapWindow ((imBLx, imBly), (imTRx, imTRy))
           ((dsBLx, dsBly), (dsTRx, dsTRY)) =
    shift (dx, dy) ∘ scale (fx, fy)
  where
    fx = (imTRx - imBLx) / (dsTRx - dsBLx)
    fy = (imTRY - imBly) / (dsTRY - dsBly)
    dx = imBLx - dsBLx * fx
    dy = imBly - dsBly * fy
```

And this is a simple utility that can be useful at times.

```
contains :: Window → Point → Bool
contains ((blx, bly), (trx, try)) (x, y) =
    (blx - x) * (x - trx) ≥ 0 &&
    (bly - y) * (y - try) ≥ 0
```

Output Pan has very snazzy support for output, including plug-ins for image processing packages and an interactive viewer. All Pancito does is write a ppm file.

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 pnm-tocmyktiff).

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
    | tot ≥ lineLimit = do hPutChar h '\n'
                          hPutStr h str
                          return len
    | otherwise       = do hPutChar h ' '
                          hPutStr h str
                          return tot
  where len = length str + 1
```

```

        tot = soFar + len

— this starts with a newline (see below)
putChunks :: Handle → [String] → IO ()
putChunks h l = do last ← foldM (putChunk h) lineLimit l
                  hPutStr h "\n"

```

...and begins with the following header:

```

— this ends needing a new line (see above)
headerPpm :: Handle → (Int, Int) → Int → IO ()
headerPpm h (x, y) n = do hPutStrLn h "P3"
                          hPutStrLn h (show x)
                          hPutStrLn h (show y)
                          hPutStr h (show n)

```

Generating Images

To print an image it must be generated: the image function must be called at each pixel and the colour converted to the correct (integer) representation. Also, because ppm does not support transparency, we need to specify a (solid) background colour.

In the code below, the transformation from pixel to image coordinates is made by a filter. The default filter (`directSample`) does a simple coordinate transformation. More sophisticated filters will be used later to implement aliasing.

```

colourStream :: Colour → (Int, Int) → Filter →
              Image → [Colour]
colourStream bg (nx, ny) f im =
  map ((+) bg ∘ f im)
    [(fromIntegral x, fromIntegral y) |
     y ← [ny,ny-1..1], x ← [1..nx]]

directSample :: (Int, Int) → Window → Filter
directSample (nx, ny) win im =
  im ∘ mapWindow win
    ((1, 1), (fromIntegral nx, fromIntegral ny))

```

Converting Colour to a series of integers is all that remains before tying everything together. Note that here, `n` is a direct scale factor and we drop the transparency (not supported by ppm).

```

colourToInts :: Int → Colour → [String]
colourToInts n (RGBA r g b _) = [flr r, flr g, flr b]
  where
    flr x = show $ max 0 (min n (floor (realToFrac n * x)))

groupRGB :: [String] → String
groupRGB s = foldr addsp "" s
  where addsp x y = x ++ " " ++ y

```

Output Routines

We now have everything necessary to write an image to a file handle. The arguments to `writePpmBase` are:

- A handle to a file that has already been opened.

- The number of bits to use in output (8, for example, means that a colour component with a value of 1.0 will be represented as 255).
- The background colour (in case the image is transparent).
- The number of pixels along the x and y axes.
- A filter that modifies the image so that when it is given integer pixel coordinates it returns appropriate colours.
- Finally, the image function itself.

```
writePpmBase :: Handle → Int → Colour → (Int, Int) →
  Filter → Image → IO ()
writePpmBase h b bg dim f im =
  do headerPpm h dim n
     putChunks h $ map (groupRGB ∘ colourToInts n) colours
  where
    n = 2b - 1
    colours = colourStream bg dim f im
```

The arguments for `writePpm` are similar to `writePpmBase`, except that you give a file name rather than an already opened handle and an appropriate filter is supplied to display a window on the image.

```
writePpm :: String → Int → Colour → (Int, Int) →
  Window → Image → IO ()
writePpm name b bg dim win im =
  do h ← openFile name WriteMode
     writePpmBase h b bg dim f im
     hClose h
  where
    f = directSample dim win
```

For examples using this routine see the tests at the end of this document.

Details

Colour By making `Colour` an instance of the `Fractional` type class we can use colours naturally in numerical expressions. `Fractional` (and its super-classes) is pretty complex, so this involves a fair amount of work.

First, functions to support operations on the components of `Colour`.

```
instance Functor Clr where
  fmap f (RGBA r g b a) = RGBA (f r) (f g) (f b) (f a)

foldC :: (a → b → a) → a → Clr b → a
foldC f z (RGBA r g b a) = f (f (f (f z r) g) b) a

zipWithC :: (a → a → b) → Clr a → Clr a → Clr b
zipWithC f (RGBA r1 g1 b1 a1) (RGBA r2 g2 b2 a2) =
  RGBA (f r1 r2) (f g1 g2) (f b1 b2) (f a1 a2)

zipC :: Clr a → Clr a → Clr (a, a)
zipC = zipWithC (,)
```

```
boolC :: (a → a → Bool) → Clr a → Clr a → Bool
boolC f a b = foldC (&&) True (zipWithC f a b)
```

```
flipC :: Clr (a, b) → (Clr a, Clr b)
flipC (RGBA (r1, r2) (g1, g2) (b1, b2) (a1, a2)) =
  ((RGBA r1 g1 b1 a1), (RGBA r2 g2 b2 a2))
```

Now we can start working through the various classes we have to satisfy.

```
instance Eq a ⇒ Eq (Clr a) where
  a == b = boolC (==) a b
```

```
instance Ord a ⇒ Ord (Clr a) where
  a ≤ b = foldC roll True (zipC a b)
  where
    roll le (x, y) = (x < y) || ((x == y) && le)
```

```
instance Show a ⇒ Show (Clr a) where
  show a = tail (foldC builds "" a)
  where builds x y = x ++ ( ' ' : show y)
```

We don't instantiate the Enum class (what does the "next" colour mean?). If you are looking for this then you probably want to use `quantisedList`.

```
delta :: Num a ⇒ Clr a
delta = RGBA 1 1 1 0
```

```
quantisedList :: Colour → Colour → Int → [Colour]
quantisedList c1 c2 n =
  fmap rescale [0 .. n-1]
  where (RGBA dr dg db da) = c2 - c1
        (RGBA r1 g1 b1 a1) = c1
        rescale x = RGBA (r1+x'*dr) (g1+x'*dg)
                          (b1+x'*db) (a1+x'*da)
        where x' = realToFrac x / realToFrac (n - 1)
```

Addition is underlay — it is not direct addition of components. So solid green + solid blue = solid blue (the second argument is on top). Subtraction is the inverse of this (peel).

If you really want to "just add" colours, use `sumComponents`.

Multiplication and division could be used in conjunction with `fromInteger` and `fromRational` to scale values. Negation produces a "photographic" negative (assuming that the colour is normalised).

```
sumComponents :: Colour → Colour → Colour
sumComponents (RGBA r1 g1 b1 a1) (RGBA r2 g2 b2 a2) =
  RGBA (r1 + r2) (g1 + g2) (b1 + b2) (a1 + a2)
```

```
instance (Fractional a, Ord a) ⇒ Num (Clr a) where
  a + b           = underlay a b
  a - b           = normalize $ peel a b
  a * b           = zipWithC (*) a b
  negate (RGBA r g b a) = RGBA (1-r) (1-g) (1-b) a
  abs a           = fmap abs a
  signum a        = fmap signum a
  fromInteger a   = fmap fromInteger (RGBA a a a a)
```

```

instance (Fractional a, Ord a) => Fractional (Clr a) where
  a / b          = zipWithC (/) a b
  recip a        = fmap recip a
  fromRational a = (RGBA b b b b) where b = fromRational a

instance (Floating a, Ord a) => Floating (Clr a) where
  pi          = (RGBA pi pi pi pi)
  exp a       = fmap exp a
  log a       = fmap log a
  sqrt a      = fmap sqrt a
  a ** b      = zipWithC (**) a b
  logBase a b = zipWithC logBase a b
  sin a       = fmap sin a
  cos a       = fmap cos a
  tan a       = fmap tan a
  asin a      = fmap asin a
  acos a      = fmap acos a
  atan a      = fmap atan a
  sinh a      = fmap sinh a
  cosh a      = fmap cosh a
  tanh a      = fmap tanh a
  asinh a     = fmap asinh a
  acosh a     = fmap acosh a
  atanh a     = fmap atanh a

```

Images It might be nice to repeat the above for Images, but I'm not that sure I could come up with suitable definitions, or whether it is possible to define instances for functions rather than data types. Instead, here are a few useful utilities:

```

addImage :: Image -> Image -> Image
addImage im1 im2 p = (im1 p) + (im2 p)

colourImage :: Colour -> Image
colourImage c _ = c

nullImage = colourImage transparent

```

Examples

Test Image All the following examples generate the same square image. On the left side is a vertical red bar, on the bottom a green bar, and in the centre a blue square. The different components have an opaqueness of 0.5 and overlap. They are displayed against a white background.

I hope that reading through these examples, and working out how they work, will explain the basics of Pancito, even if you are fairly new to Haskell. Good luck!

Basics This is a very simple, direct implementation of the test image.

```

exImage1 :: Image
exImage1 (x, y) = r + g + b
  where
    r | bar x y = RGBA 1 0 0 0.5

```

```

    | otherwise = transparent
g | bar y x    = RGBA 0 1 0 0.5
  | otherwise = transparent
b | square x y = RGBA 0 0 1 0.5
  | otherwise = transparent
bar p q      = p>0.05 && p<0.2 && q>0.1 && q<0.95
square p q   = p>0.15 && p<0.7 && q>0.15 && q<0.7

doEx1 :: IO ()
doEx1 = writePpm "ex1.ppm" 8 white (20, 20) square01 exImage1

```

Images A different way of writing this would be to combine three different images:

```

bar :: Point → Bool
bar (p, q) = p>0.05 && p<0.2 && q>0.1 && q<0.95

square :: Point → Bool
square (p, q) = p>0.15 && p<0.7 && q>0.15 && q<0.7

exRImage2, exGImage2, exBImage2 :: Image
exRImage2 p =
  if bar p then opacity 0.5 red else transparent
exGImage2 (x, y) =
  if bar (y, x) then opacity 0.5 green else transparent
exBImage2 p =
  if square p then opacity 0.5 blue else transparent

exImage2 :: Image
exImage2 = foldl addImage exRImage2 [exGImage2, exBImage2]

doEx2 :: IO ()
doEx2 = writePpm "ex2.ppm" 8 white (20, 20) square01 exImage2

```

Maps And yet another approach is to make the three images mappings of a unit square:

```

squareImage :: Colour → Image
squareImage c p = if contains square01 p
                  then c
                  else transparent

exRImage3, exGImage3, exBImage3 :: Image
exRImage3 = squareImage (opacity 0.5 red) ◦
            mapWindow square01 ((0.05,0.1),(0.2,0.95))
exGImage3 = squareImage (opacity 0.5 green) ◦
            mapWindow square01 ((0.1,0.05),(0.95,0.2))
exBImage3 = squareImage (opacity 0.5 blue) ◦
            mapWindow square01 ((0.15,0.15),(0.7,0.7))

exImage3 :: Image
exImage3 = foldl addImage exRImage3 [exGImage3, exBImage3]

doEx3 :: IO ()
doEx3 = writePpm "ex3.ppm" 8 white (20, 20) square01 exImage3

```

Display You could execute one of these examples with a file like this (which I'll call `ex1.hs`; this is not Pancito source code).

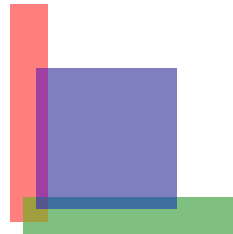
```
module Main where
import Pancito
main = doEx1
```

For `ghc` you can then generate and display the result by typing

```
ghc -c Pancito.lhs ; ghc ex1.hs Pancito.o
./a.out
xloadimage ex1.ppm
```

Alternatively, for postscript:

```
ghc -c Pancito.lhs
ghc ex1.hs Pancito.o
./a.out
pnmtops -scale 5 ex1.ppm > ex1.ps
```



Utilities This section includes code that I have found useful while making images. The functions here are not used in any of the earlier code.

If you send me useful functions I'll include them here (if I like them too!).

Windowed Images Sometimes it's nice to start with an image defined by a window.

```
windowedImage :: Window → Colour → Colour → Image
windowedImage w c1 c2 p = if contains w p then c1 else c2
```

Polar Coordinates These are lifted directly from Pan — they make it easy to handle circular images and can also simplify other calculations.

I haven't defined a new type (it's just an alias, not a "real" type) for polar coordinates, so be careful that you don't mix them with the normal system unintentionally. All conversions must be explicit.

```
type Polar = Point

distSq, dist, angle :: Point → Double
distSq (x, y) = x^2 + y^2
dist = sqrt ∘ distSq
angle (x, y) = atan2 y x

fromPolar :: Polar → Point
fromPolar (r, t) = (r * cos t, r * sin t)

toPolar :: Point → Polar
toPolar p = (dist p, angle p)
```

Comparing Colours As colours are floating point values, it is unlikely that even apparently equal colours will be identical.

```
similar :: Double -> Colour -> Colour -> Bool
similar d = boolC close
  where close c1 c2 = abs (c1 - c2) <= d
```

Aliasing High contrast diagonal lines can show jagged edges if an image has large pixels. Often the size of the pixels cannot be changed, but the appearance can be improved by generating a higher resolution image and then averaging the result back down to the required resolution.

Because this involves calculating an intermediate, higher resolution image, it takes proportionally longer to generate (2x2 pixel antialiasing takes four times longer than the direct image, for example).

AntiAlias is a good example of a complex filter. It converts an image so that when supplied by a single point the colour is evaluated on a grid of $n \times n$ points, averaged and returned. Both the input point to the enclosed image and the return value from the image are modified.

```
antiAlias :: Int -> (Int, Int) -> Window -> Filter
antiAlias n (nx, ny) win im p =
  averageColour n $ map im' $ mkPattern n p
  where
    im' = im o mapWindow win
          ((1, 1), (fromIntegral nx, fromIntegral ny))
```

```
mkPattern :: Int -> Point -> [Point]
mkPattern n (x, y) =
  map shfsc1 [(fromIntegral x', fromIntegral y') |
             x' <- [1..n], y' <- [1..n]]
  where
    shfsc1 (u, v) = (x + shfsc1' u, y + shfsc1' v)
    shfsc1' z = (z - 0.5) / (fromIntegral n) - 0.5
```

```
averageColour :: Int -> [Colour] -> Colour
averageColour n l =
  (foldl sumComponents transparent 1) / (fromIntegral n**2)
```

Wrap all this up to provide a way of generating aliased images:

```
writePpmAlias :: Int -> String -> Int -> Colour -> (Int, Int) ->
  Window -> Image -> IO ()
writePpmAlias n name b bg dim win im =
  do h <- openFile name WriteMode
     writePpmBase h b bg dim f im
     hClose h
  where
    f = antiAlias n dim win
```

Hints These are suggestions for how to use this package.

Small Images Develop ideas with small images. Once everything is working it is easy to change the number of pixels in the output to get the size/resolution you want. The time taken increase with the number of pixels, so doubling

the length and height of an image increases the time required by a factor of four.

Use a Compiler Use a compiler (like `ghc`) for compiling Haskell to generate images. It will produce images more quickly than an interpreter.

Use an Interpreter Use an interpreter (like `hugs`) for debugging programs. You can load your files and then run individual functions.

If you are used to debugging programs by adding print statements you'll find Haskell difficult at first — because it is a pure functional program you cannot add print statements without introducing the IO monad, which usually involves a lot of work. But once you start testing functions in `hugs` you'll find that it is an efficient way of finding errors (the functional purity helps make this simple; you don't need to worry about side effects etc).

Don't Alias Don't alias images during development. Add aliasing at the end if you need to remove jaggies, but avoid wasting time with it when you are exploring new ideas.

Monitor Note that `xloadimage` will display incomplete ppm images, so you can monitor the progress of large (slow) images as they are being generated.

Documentation You can generate a postscript document from this file by typing:

```
latex Pancito.lhs
dvips Pancito -o Pancito.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).

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

Licensing Conditions This document and code are released under the GPL (see www.gnu.org for full details). However, I also ask that it is not used by Israeli citizens living in Israel (unless, of course, there is lasting peace in the area).

Copyright 2001 Andrew Cooke (Jara Software).