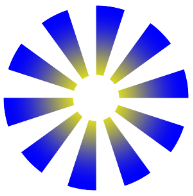


Functional Images

CONAL ELLIOTT
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
`http://research.microsoft.com/~conal`

Abstract

There have been many libraries for generating images using functional programming, each based on its own fixed set of geometric primitives and combinators. In contrast, this paper addresses the general notion of images, based on a very simple model: functions over continuous 2D space. Spatial transformations are mappings from 2D space to itself, and regions are just Boolean-valued images. This basis suffices for giving elegant expression to a wide range of images, as we illustrate through many examples. The library *Pan* embodying these ideas is freely available, and we hope to inspire others to join in the delightful search for new visual building blocks.

1 Introduction

Many of us have been drawn to functional languages from a sense of beauty of expression. It is a joy to express our ideas with simplicity and generality and then compose them in endless variety. Software used to produce visual beauty, on the other hand, is usually created with imperative languages and generally lacks the sort of "inner beauty" that we value. When occasionally one gets to combine these two kinds of beauty, the process, and sometimes the result, is a great pleasure. This paper describes one such combination in an attempt to share this pleasure and inspire others to join in the exploration.

Computer-generated images are often constructed from an underlying "geometric" model, composed of lines, curves, polygons in 2D, or illuminated and textured curved or polyhedral surfaces in 3D. Just as images are often presentations of geometric models, so also are geometric models often presentations of more specialized or abstract models, such as text (presented via outline fonts) or financial data (presented via pie charts).

The distinction between geometry and image, and more generally, between model and presentation (Elliott, 1999), is very valuable, in that it allows one to concentrate on the underlying model and rely on a library to take care of presentation. In doing so, it becomes easier and easier to describe fewer and fewer images, but what about the general notion of "image"?

Functional languages are particularly good at the model-oriented approach to image generation, thanks to their excellent support for modularity. There does not, however, seem to be much work done in functional programming on supporting the general notion of images. This oversight is puzzling, because images can be modeled very naturally *as functions* from a 2D domain to colors. This formulation is especially elegant when the 2D domain is continuous, non-rectangular and possibly of infinite extent. Adding another dimension for (continuous) time is just as easy, yielding temporally and spatially scalable image-based animation.

We have explored this very simple notion of images as functions in a Haskell library–a "domain-specific embedded language" (DSEL) (Hudak, 1998)–that we call *Pan*. This paper presents the types and operations that make up Pan, and illustrates their use through a collection of examples. Some of the examples are synthesized from mathematical descriptions, while others are image-transforming "filters" that can be applied to photographs or synthetic images. For more examples, including color and animations, see the example gallery and the web version of this paper, both available through the Pan web page, where Pan is freely available for downloading.[1]

As is often the case with DSELs, some properties of the functional host language turn out to be quite useful in practice. Firstly, higher-order functions are essential, since images are functions. Parametric polymorphism allows images whose "pixels" are of any type at all, with some image operations being polymorphic over pixel type. Aside from color-valued images, Boolean images can serve as a general notion of "regions" for image masking or selection, and real-valued images can represent 3D height fields or spatially-varying parameters for color generation. Dually, some operations are polymorphic in the domain rather than the range type. These operations might be used to construct "solid textures", which are used in 3D graphics to give realistic appearance to simulated clouds, stone and wood. So far, we have not needed laziness, so a translation to Standard ML should be straightforward and satisfactory.

For efficiency, Pan is implemented as a compiler. It fuses the code fragments used in constructing an image as well as the display function itself, performs algebraic simplification, common-subexpression elimination and code hoisting, and produces C code, which is then given to an optimizing compiler (Elliott *et al.*, 2001).

The contributions of this paper are as follows:

- We propose a strikingly simple but precise model for resolution-independent images of any type that fits neatly into modern typed functional languages.
- Within this model, we give precise and simple definitions for a library of useful image building blocks.
- Through a variety of examples, we demonstrate that the simple model is capable of producing a range of visually interesting images.

[1] See `http://research.microsoft.com/~conal/pan`. For now, running the Pan compiler requires having the Microsoft C++ compiler.

2 What is an image?

Pan's model of images is simply functions from infinite, continuous 2D space to colors with partial opacity. (Although the domain space is infinite, some images are transparent everywhere outside of a bounding region.) One might express the definition of images as follows:[2]

$$\textbf{type}\ \mathit{Image} = \mathit{Point} \to \mathit{Color} \quad \text{— first try}$$

where

$$\textbf{type}\ \mathit{Point} = (\mathit{Float}, \mathit{Float}) \quad \text{— Cartesian coords}$$

It is useful, however, to generalize the semantic model of images so that the range of an image is not necessarily *Color*, but an arbitrary type. For this reason, *Image* is really a type *constructor*:

$$\textbf{type}\ \mathit{Image}\ c = \mathit{Point} \to c$$

It can also be useful to generalize the domain of images, from points in 2D space to other types (such as 3D space or points with integer coordinates), but we shall not exploit that generality in this paper.

Boolean-valued "images" are useful for representing arbitrarily complex spatial regions (or "point sets") for complex image masking. This interpretation is just the usual identification between sets and characteristic functions:

$$\textbf{type}\ \mathit{Region} = \mathit{Image}\ \mathit{Bool}$$

As a first example, Figure 1 shows an infinitely tall vertical strip of unit width, *vstrip*, as defined below.[3]

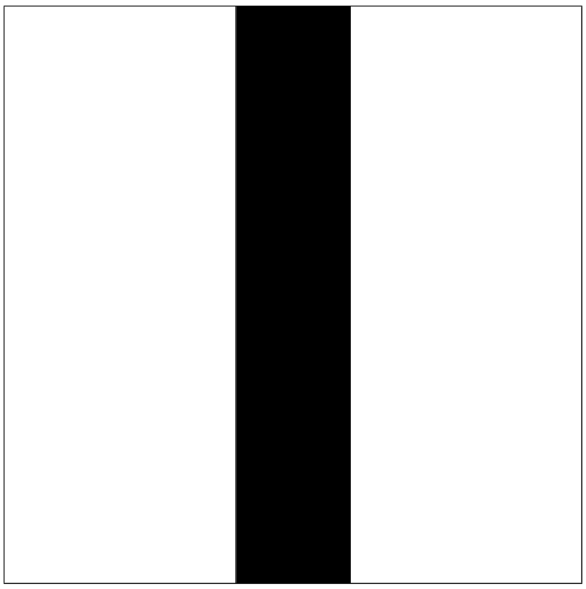
$$\begin{aligned}&\mathit{vstrip} :: \mathit{Region}\\ &\mathit{vstrip}\ (x, y) = |x| \leq 1/2\end{aligned}$$

For a slightly more complex example, consider the checkered region shown in Figure 2. The trick is to take the floor of the pixel coordinates and test whether the sum is even or odd. Whenever x or y passes an integer value, the parity of $x + y$ changes.

$$\begin{aligned}&\mathit{checker} :: \mathit{Region}\\ &\mathit{checker}\ (x, y) = \mathit{even}\ (\lfloor x \rfloor + \lfloor y \rfloor)\end{aligned}$$

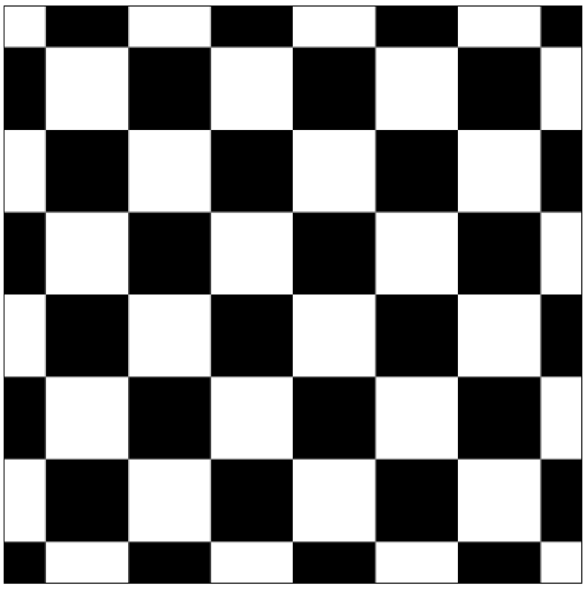
[2] All definitions in this paper are expressed in Haskell (Jones *et al.*, 1999). We take some small liberties with notation. As described elsewhere (Elliott *et al.*, 2001), our implementation really uses "expression types" with names like *FloatE* instead of *Float*, in order to optimize and compile Pan programs into efficient machine code. Operators and functions are overloaded to work on expression types where necessary, but a few require special names, such as "$==*$" and "*notE*". The definitions used in this paper could, however, be used directly as a valid but less efficient implementation. We also use some standard math notation for functions like absolute value, floor, and square root.

[3] Each figure shows an origin-centered finite window onto an infinite image and is annotated with the width of the window in logical coordinates. For instance, this figure shows the window $[-7/2, 7/2] \times [-7/2, 7/2]$ onto the infinite *vstrip* image.



$[width = 7]$

Fig. 1. *vstrip*



$[width = 7]$

Fig. 2. *checker*

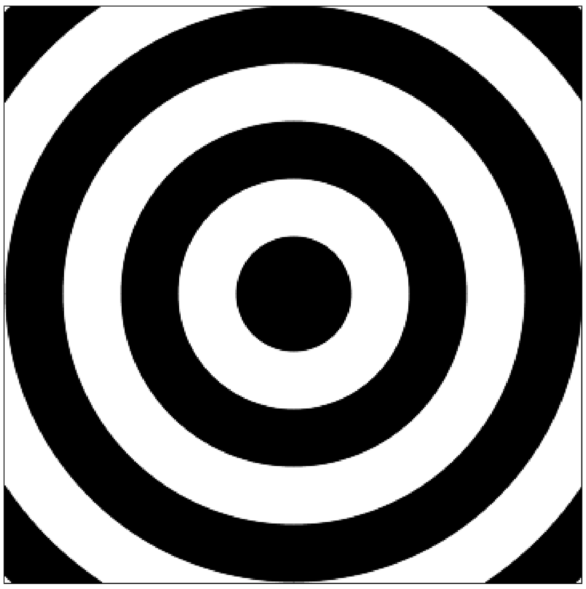


Fig. 3. *altRings*

Images need not have straight edges and right angles. Figure 3 shows a collection of concentric black&white rings. The definition is similar to *checker*, but uses the distance from the origin to a given point, as computed by *distO*.

$$altRings\ p\ =\ even\ \lfloor distO\ p \rfloor$$

The distance-to-origin function is also easy to define:

$$distO\ (x, y)\ =\ \sqrt{x^2 + y^2}$$

It is often more convenient to define images using polar coordinates (ρ, θ) rather than rectangular coordinates (x, y). The following definitions are helpful.

$$type\ PolarPoint\ =\ (Float,\ Float)$$

$$\begin{aligned}
&fromPolar :: Point \rightarrow PolarPoint\\
&toPolar\ \ \ \ :: PolarPoint \rightarrow Point\\
&fromPolar(\rho, \theta) = (\rho * cos\ \theta,\ \rho * sin\ \theta)\\
&toPolar\ \ \ \ (x, y) = (distO\ (x, y),\ atan2\ y\ x)
\end{aligned}$$

Figure 4 shows a "polar checkerboard", defined using polar coordinates. The integer parameter n determines the number of alternations, and hence is twice the number of slices. (We will see a simpler definition of *polarChecker* in Section 7.)

$$\begin{aligned}
&polarChecker :: Int \rightarrow Region\\
&polarChecker\ n\ =\ checker \circ sc \circ toPolar
\end{aligned}$$

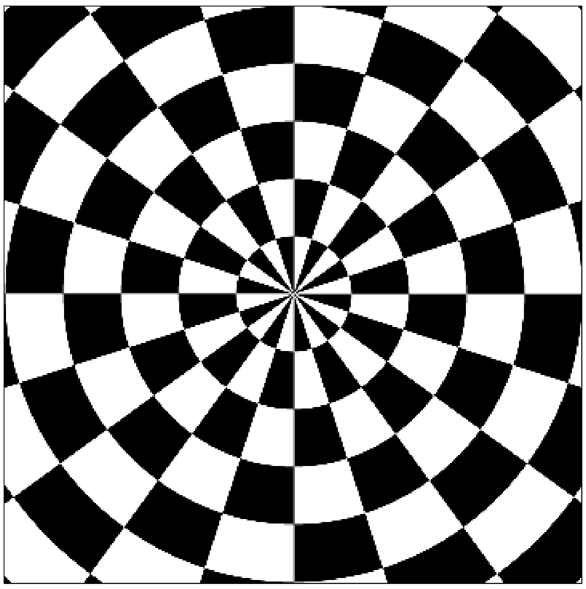


Fig. 4. *polarChecker* 10

where
$sc\,(\rho,\theta) = (\rho,\ \theta * n'/\pi)$
$n' = \mathit{fromInt}\ n$

For a different sort of example, the following simple definition describes a version of the famous Sierpinski Gasket (Figure 5) (Mandelbrot, 1977). This formulation describes the region as the set of points (x, y) for which $\lfloor y \rfloor$ contains no one-bits beyond those present in $\lfloor x \rfloor$, making this comparison using bitwise *or* ("$.|.$"):[4]

$\mathit{gasket} :: \mathit{Region}$
$\mathit{gasket}\,(x,y) = \lfloor x \rfloor \mathbin{.|.} \lfloor y \rfloor == \lfloor x \rfloor$

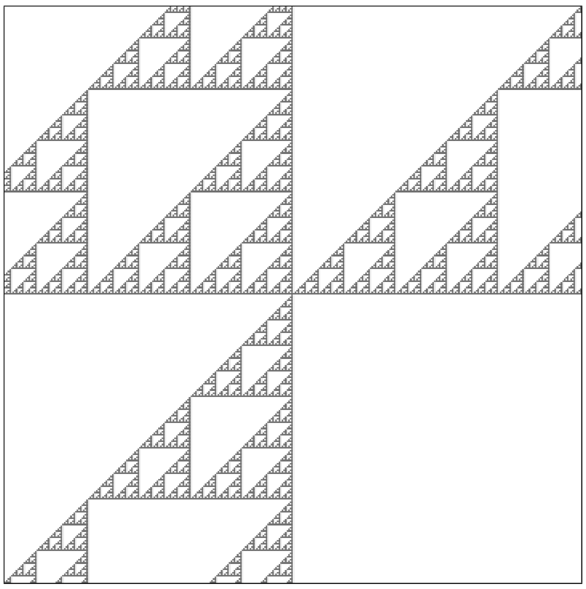
For grey-scale images, we can use as "pixel" values within the real interval $[0,1]$. This constraint is not expressible in the type system of our language, but as a reminder, we introduce the type synonym *Frac*:

type *Frac* = *Float* — In $[0,1]$

Figure 6 shows a wavy grey-scale image that shifts smoothly between white (zero) and black (one) in concentric rings.

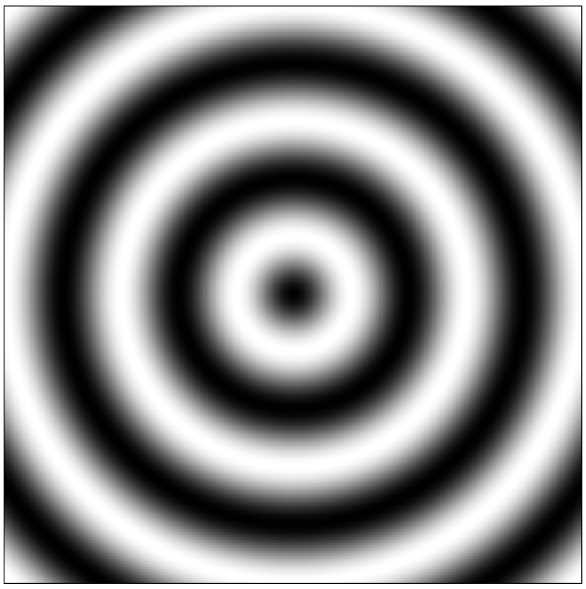
$\mathit{wavDist} :: \mathit{Image\ Frac}$
$\mathit{wavDist}\ p = (1 + \cos\,(\pi * \mathit{distO}\ p))\ /\ 2$

[4] Thanks to Craig Kaplan for suggesting the use of bitwise *or* and Sigbjørn Finne for refining the original definition.



[*width* = 1440]

Fig. 5. *gasket*



[*width* = 10]

Fig. 6. *wavDist*

3 Colors

Pan colors are quadruples of real numbers in $[0,1]$, with the first three components for blue, green, and red (BGR) components, and the last for transparency ("alpha"):

$$\textbf{type}\ \mathit{Color} = (\mathit{Frac}, \mathit{Frac}, \mathit{Frac}, \mathit{Frac}) \quad \text{— BGRA}$$

The blue, green, and red components will have alpha multiplied in already, and so must less be than or equal to alpha (i.e., we are using "pre-multiplied alpha" (Smith, 1995)). Given this constraint, there is exactly one fully transparent color:

$$\mathit{invisible} = (0, 0, 0, 0)$$

We are now in a position to define some familiar (completely opaque) colors:

$$\begin{aligned} &\mathit{red} &&= (0, 0, 1, 1) \\ &\mathit{green} &&= (0, 1, 0, 1) \\ &\ldots \end{aligned}$$

It is often useful to linearly interpolate ("lerp") between colors, to create a smooth transition through space or time. This is the purpose of *lerpC* $w\ c_1\ c_2$. The first parameter w is a fraction, indicating the relative weight of the color c_1. The weight assigned to the second color c_2 is $1 - w$:

$$\begin{aligned} &\mathit{lerpC} :: \mathit{Frac} \to \mathit{Color} \to \mathit{Color} \to \mathit{Color} \\ &\mathit{lerpC}\ w\ (b_1, g_1, r_1, a_1)\ (b_2, g_2, r_2, a_2) = (h\ b_1\ b_2, h\ g_1\ g_2, h\ r_1\ r_2, h\ a_1\ a_2) \\ &\quad \textbf{where} \\ &\qquad h\ x_1\ x_2 = w * x_1 + (1 - w) * x_2 \end{aligned}$$

With *lerpC*, we can define other useful functions, e.g.,

$$\begin{aligned} &\mathit{lighten}, \mathit{darken} :: \mathit{Fraction} \to \mathit{Color} \to \mathit{Color} \\ &\mathit{lightenx}\ c = \mathit{lerpC}\ x\ c\ \mathit{white} \\ &\mathit{darkenx}\ c = \mathit{lerpC}\ x\ c\ \mathit{black} \end{aligned}$$

It is also easy to extend color interpolation to two dimensions, by making three applications of linear interpolation–two horizontal and one vertical (or, equivalently, two vertical and one horizontal). Figure 7 illustrates this operation, and is centered at $(1/2, 1/2)$ rather than the origin. Black, red, blue and white are the colors in the four corners. Note the partial application of *bilerpC* to four arguments, resulting in an image, which is a function, though one expected to be sampled on a finite region.

$$\mathit{bilerpBRBW} = \mathit{bilerpC}\ \mathit{black}\ \mathit{red}\ \mathit{blue}\ \mathit{white}$$

$$\begin{aligned} &\mathit{bilerpC} :: \mathit{Color} \to \mathit{Color} \to \mathit{Color} \to \mathit{Color} \to (\mathit{Frac}, \mathit{Frac}) \to \mathit{Color} \\ &\mathit{bilerpC}\ \mathit{ll}\ \mathit{lr}\ \mathit{ul}\ \mathit{ur}\ (\mathit{wx}, \mathit{wy}) = \mathit{lerpC}\ \mathit{wy}\ (\mathit{lerpC}\ \mathit{wx}\ \mathit{ll}\ \mathit{lr})\ (\mathit{lerpC}\ \mathit{wx}\ \mathit{ul}\ \mathit{ur}) \end{aligned}$$

Because of the type invariant on colors, this definition only makes sense if *wx* and *wy* fall in the interval $[0,1]$.

An operation similar to *lerpC* is color overlay, which will be used in the next

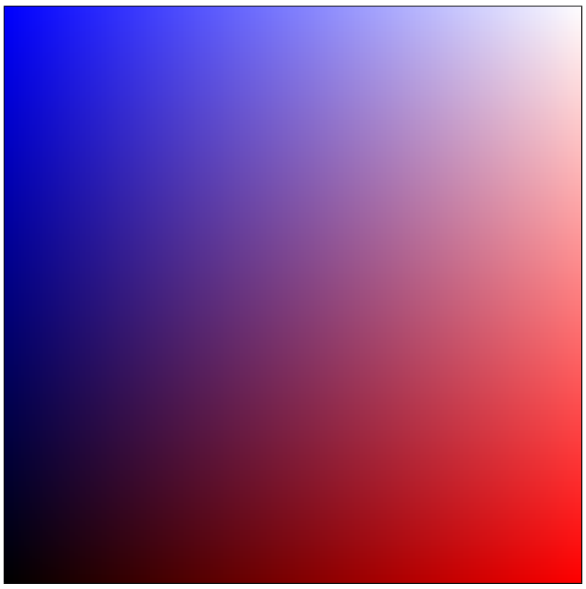


Fig. 7. *bilerpBRBW*

section to define image overlay. The result is a blend of the two colors, depending on the opacity of the top (first) color. A full discussion of this definition can be found in Smith (1995):

$$
\begin{aligned}
&cOver :: Color \to Color \to Color\\
&cOver\,(b_1,\, g_1,\, r_1,\, a_1)\,(b_2,\, g_2,\, r_2,\, a_2) = (h\; b_1\; b_2,\; h\; g_1\; g_2,\; h\; r_1\; r_2,\; h\; a_1\; a_2)\\
&\quad \textbf{where}\\
&\qquad h\; x_1\; x_2 = x_1 + (1 - a_1) * x_2
\end{aligned}
$$

Not surprisingly, color-valued images are of particular interest, so we'll use a convenient abbreviation:

$$\textbf{type}\; ImageC = Image\; Color$$

4 Pointwise lifting

Many image operations result from pointwise application of operations on one or more values. For example, the overlay of one image on top of another can be defined in terms of *cOver* :

$$
\begin{aligned}
&over :: ImageC \to ImageC \to ImageC\\
&(top\; `over`\; bot)\; p = top\; p\; `cOver`\; bot\; p
\end{aligned}
$$

This commonly arising pattern is supported by a family of "lifting" functionals:[5]

$$
\begin{array}{l}
\mathit{lift}_1 :: (a \to b) \qquad\qquad\quad\;\; \to (p \to a) \to (p \to b) \\
\mathit{lift}_2 :: (a \to b \to c) \qquad\quad\;\; \to (p \to a) \to (p \to b) \to (p \to c) \\
\mathit{lift}_3 :: (a \to b \to c \to d) \to (p \to a) \to (p \to b) \to (p \to c) \to (p \to d) \\
\cdots
\end{array}
$$

$$
\begin{array}{ll}
\mathit{lift}_1\ h\ f_1 & p = h\ (f_1\ p) \\
\mathit{lift}_2\ h\ f_1\ f_2 & p = h\ (f_1\ p)\ (f_2\ p) \\
\mathit{lift}_3\ h\ f_1\ f_2\ f_3 & p = h\ (f_1\ p)\ (f_2\ p)\ (f_3\ p) \\
\cdots &
\end{array}
$$

Then $\mathit{over} = \mathit{lift}_2\ \mathit{cOver}$.

Other examples of pointwise lifting includes selection (*cond*) and interpolation (*lerpI*) between two images:[6]

$$
\begin{array}{l}
\mathit{cond} :: \mathit{Image}\ \mathit{Bool} \to \mathit{Image}\ c \to \mathit{Image}\ c \to \mathit{Image}\ c \\
\mathit{cond} = \mathit{lift}_3\ (\lambda\ a\ b\ c \to \mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c)
\end{array}
$$

$$
\begin{array}{l}
\mathit{lerpI} :: \mathit{Image}\ \mathit{Frac} \to \mathit{ImageC} \to \mathit{ImageC} \to \mathit{ImageC} \\
\mathit{lerpI} = \mathit{lift}_3\ \mathit{lerpC}
\end{array}
$$

Zero-ary lifting is already provided by Haskell's *const* function:

$$
\begin{array}{l}
\mathit{const} :: a \to (p \to a) \\
\mathit{const}\ a\ p = a
\end{array}
$$

Given *const*, we can define the empty image and give convenient names to several opaque, constant-color images:

$$
\begin{array}{ll}
\mathit{empty} & = \mathit{const}\ \mathit{invisible} \\
\mathit{whiteI} & = \mathit{const}\ \mathit{white} \\
\mathit{blackI} & = \mathit{const}\ \mathit{black} \\
\mathit{redI} & = \mathit{const}\ \mathit{red} \\
\cdots &
\end{array}
$$

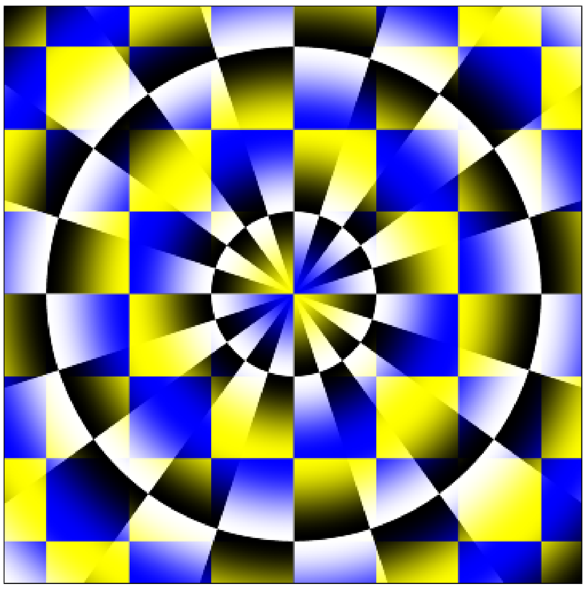
Note that *all* pointwise-lifted functions are polymorphic over the domain type (not necessarily *Point*), and so could work for 1D images (e.g., interpreted as sound), 3D images (sometimes called "solid textures"), or ones over discrete or abstract domains as well.

Figure 8 shows a simple example of image interpolation based on the examples in Figures 6, 2, and 4. Since *lerpI* works on color images, we must first color the region arguments.

$$\mathit{blackWhiteIm}, \mathit{blueYellowIm} :: \mathit{Region} \to \mathit{ImageC}$$

[5] For intuition, think of p as *Point*, so that $p \to a = \mathit{Image}\ a$ and similarly for b, c, d. These lifting functionals special cases of the Haskell monadic lifting functionals, applied to the reader monad.

[6] In a call-by-value language, *cond* would need to be defined differently, in order to avoid unnecessary evaluation.



[$width = 7$]

Fig. 8.
$lerpI\ \ wavDist(blackWhiteIm\ (polarChecker\ 10))$
$(blueYellowIm\ \ checker)$

$$\begin{aligned} blackWhiteIm\ reg &= \ cond\ reg\ blackI\ whiteI \\ blueYellowIm\ reg &= \ cond\ reg\ blueI\ yellowI \end{aligned}$$

As a simpler example, Figure 9 interpolates between blue and yellow, and will be useful in several later examples.

$$ybRings \ = \ lerpI\ wavDist\ blueI\ yellowI$$

5 Spatial Transformations

In computer graphics, spatial transformationss are typically represented by matrices, and hence are restricted to special classes like linear, affine, or projective. Application of transformations is implemented as a matrix/vector multiplication, and composition as matrix/matrix multiplication. In fact, this representation is so common that spatial transformations are often thought of as *being* matrices. A simpler and much more general point of view, however, is that they are simply space-to-space functions.

$$\textbf{type}\ Warp \ = \ Point \ \rightarrow \ Point$$

It is then easy to define the familiar affine warps:

$$\textbf{type}\ Vector \ = \ (Float, Float)$$

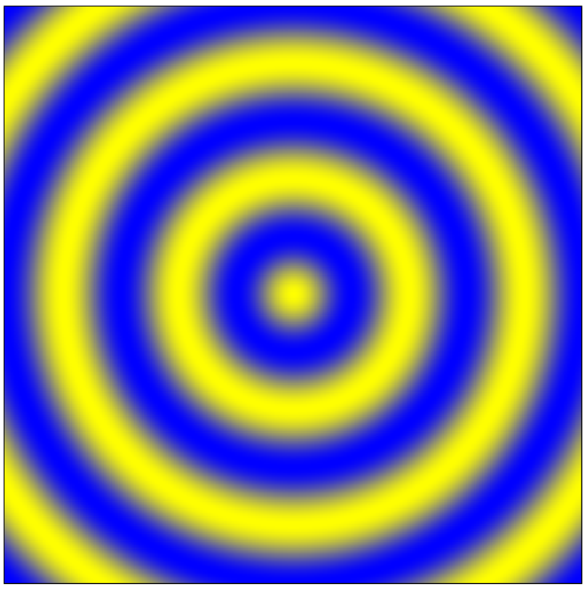


Fig. 9. *ybRings*

$$\begin{array}{l} \mathit{translateP} :: \mathit{Vector} \rightarrow \mathit{Warp} \\ \mathit{translateP}\,(dx,\ dy)\,(x,y) = (x + dx,\ y + dy) \end{array}$$

$$\begin{array}{l} \mathit{scaleP} :: \mathit{Vector} \rightarrow \mathit{Warp} \\ \mathit{scaleP}\,(sx,\ sy)\,(x,y) = (sx * x,\ sy * y) \end{array}$$

$$\begin{array}{l} \mathit{uscaleP} :: \mathit{Float} \rightarrow \mathit{Warp} \quad \text{— uniform} \\ \mathit{uscaleP}\ s = \mathit{scaleP}\,(s,s) \end{array}$$

$$\begin{array}{l} \mathit{rotateP} :: \mathit{Float} \rightarrow \mathit{Warp} \\ \mathit{rotateP}\,\theta\,(x,y) = (x * \cos\theta - y * \sin\theta, y * \cos\theta + x * \sin\theta\ ) \end{array}$$

By definition, warps map points to points. Can we "apply" them, in some sense, to map images into warped images?

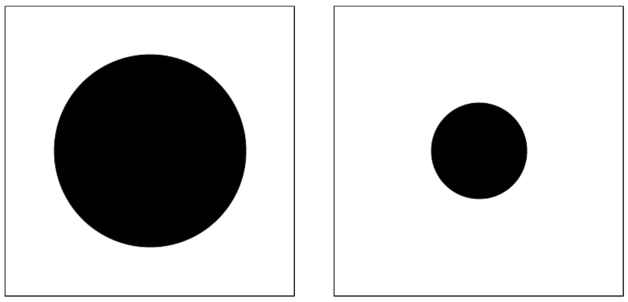
$$\mathit{applyWarp} :: \mathit{Warp} \rightarrow \mathit{Image}\ c \rightarrow \mathit{Image}\ c$$

A look at the definitions of the *Image* and *Warp* types suggests the following simple definition:

$$\mathit{applyWarp}\ \mathit{warp}\ \mathit{im} \overset{?}{=} \mathit{im} \circ \mathit{warp} \quad \text{— } \textbf{wrong}$$

Figure 10 shows a unit disk *udisk* and the result of $\mathit{udisk} \circ \mathit{uscaleP}\ 2$, where

$$\begin{array}{l} \mathit{udisk} :: \mathit{Region} \\ \mathit{udisk}\ p = \mathit{distO}\ p < 1 \end{array}$$



[*width* = 3]

Fig. 10. Disk *udisk* (left), and *udisk* ∘ *uscaleP* 2 (right)

Notice that the *uscaleP*-composed *udisk* is *half* rather than twice the size of *udisk*. (Similarly, *udisk* ∘ *translateP* $(1, 0)$ moves *udisk* to the *left* rather than right.) The reason is that *uscaleP* 2 maps input points to be twice as far from the origin, so points have to start out within $1/2$ unit of the origin in order for their scaled counterparts to be within 1 unit.

In general, to warp an image, we must *inversely* warp sample points before feeding them to the image being warped:

$$applyWarp\ warp\ im\ =\ im\ \circ\ inverse\ warp$$

While this definition is simple and general, it has the serious problem of requiring inversion of arbitrary spatial mappings. Not only is it sometimes difficult to construct inverses, but also some interesting mappings are many-to-one and hence not invertible. In fact, from an image-centric point-of-view, we *only* need the inverses and not the warps themselves. For these reasons, we simply construct the warps in inverted form.[7] The “wrong” version of *applyWarp* then becomes right if used with inverses, so we’ll rename it:

$$invWarp\ warp\ im\ =\ im\ \circ\ warp$$

Because it can be mentally cumbersome always to think of warps as functions and warp-application as composition, Pan provides a friendly vocabulary of image-warping functions:

```
type Filter c = Image c → Image c

translate, scale :: Vector → Filter c
uscale, rotate   :: Float  → Filter c
```

[7] Easy invertibility is one of the benefits of restricting warps to be affine and representing them as matrices. A middle ground, offering invertibility and reasonable generality, would be to represent warps as inverse pairs of functions. Composition could be defined as *compose* $(f, f')\ (g, g') = (f \circ g, g' \circ f')$.

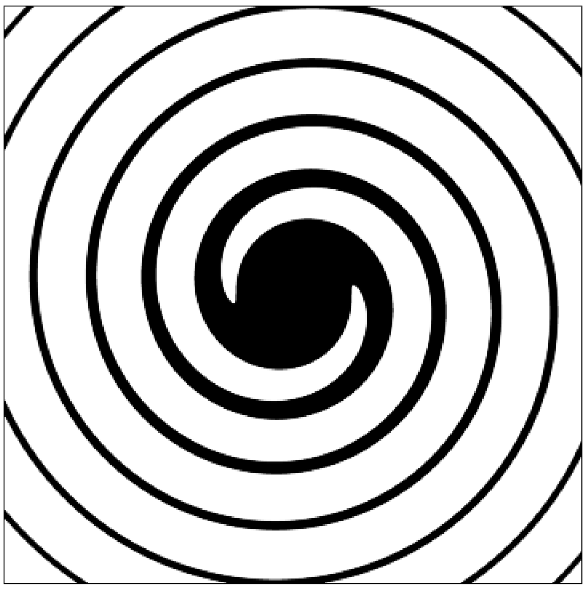


Fig. 11. *swirl* 1 *vstrip*

$$\begin{array}{llll}
\textit{translate}(dx, dy) & = \textit{invWarp}\,(\textit{translateP} & (-dx,\ -dy)) \\
\textit{scale}\quad (sx, sy) & = \textit{invWarp}\,(\textit{scaleP} & (1/sx,\ 1/sy)) \\
\textit{uscale}\quad s & = \textit{invWarp}\,(\textit{uscaleP} & (1/s)) \\
\textit{rotate}\quad \theta & = \textit{invWarp}\,(\textit{rotateP} & (-\theta))
\end{array}$$

In addition to these familiar affine warps, one can define any other kind of space-to-space function, limited only by one's imagination. For instance, here is a "swirling" warp. It takes each point p and rotates it about the origin by an amount that depends on the distance from p to the origin. For predictability, this warp takes a parameter r that gives the distance at which a point is rotated through a complete circle (2π radians):

```
swirlP :: Float → Warp
swirlP r p = rotate (distO p ∗ (2 π / r)) p

swirl :: Float → Filter c    — Image version
swirl r = invWarp (swirlP (−r))
```

Applying the *swirl* effect to *vstrip* (Figure 1) defined earlier results in an infinite spiral whose arms thin out away from the origin (Figure 11).

6 Animation

Just as an image is a function of space, an animation is a function of continuous time.

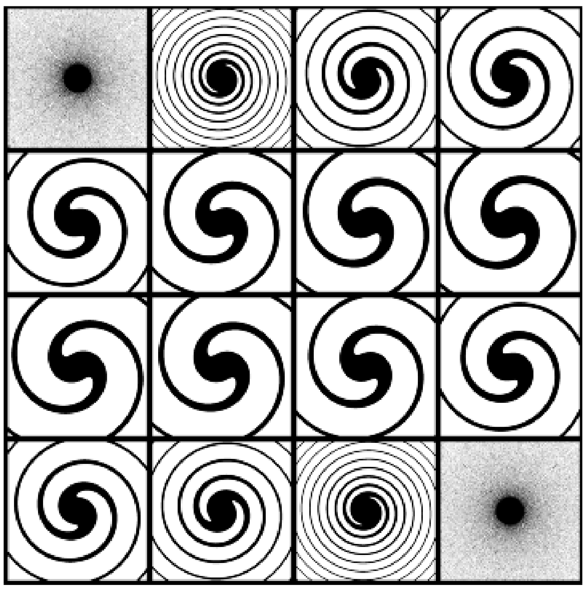


Fig. 12. *swirlingVStrip*

$$\textbf{type}\ Time = Float$$
$$\textbf{type}\ Anim\ c = Time \rightarrow Image\ c$$

This model is adopted from Fran (Elliott & Hudak, 1997; Elliott, 1999), and leads to temporal resolution independence, which allows animations to be warped in time as easily as images are warped in space.

As a simple animation example, let the "swirl factor" in Figure 1 vary with time, say between -3 and 3 (Figure 12).

$$swirlingVStrip :: Anim\ Bool$$
$$swirlingVStrip\ t = swirl\ (3 * sin\ t)\ vstrip$$

Figure 13 shows what *swirl* does to the half plane *xPos* given by $x > 0$. We square time to emphasize small and large values of the *swirl* parameter.

$$swirlingXPos :: Anim\ Bool$$
$$swirlingXPos\ t = swirl\ (t^2)\ xPos$$

$$xPos :: Region$$
$$xPos\,(x, y) = x > 0$$

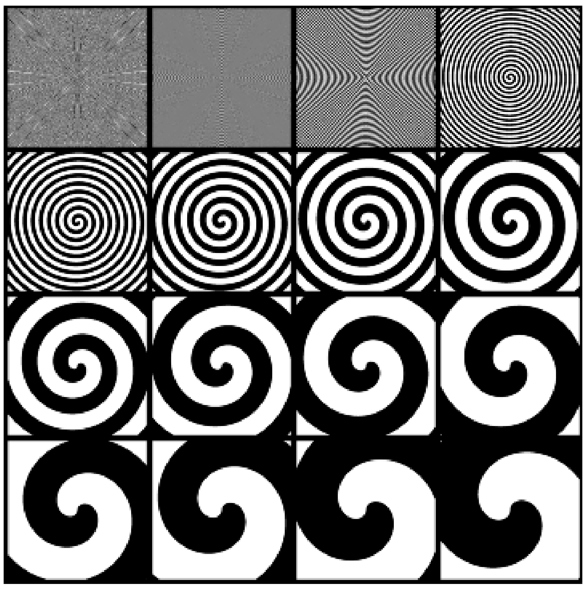


Fig. 13. *swirlingXPos*

7 Region algebra

Boolean images are useful in many situations, and can be thought of as "regions" of space. This interpretation is just the usual identification between sets and characteristic functions:

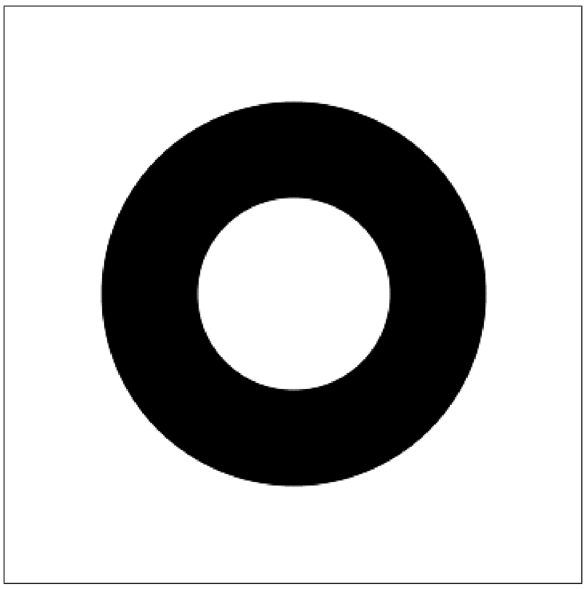
$$\textbf{type}\ Region = Image\ Bool$$

Set operations are useful and easy to define:

$$\begin{aligned}
&(\cap), (\cup), xorR, (\backslash) :: Region \to Region \to Region\\
&compR :: Region \to Region\\
&universeR, emptyR :: Region
\end{aligned}$$

$$\begin{aligned}
(\cap) &= lift_2\ (\wedge)\\
(\cup) &= lift_2\ (\vee)\\
xorR &= lift_1\ xor\\
compR &= lift_1\ not\\
universeR &= const\ True\\
emptyR &= const\ False\\
r \setminus r' &= r \cap compR\ r'
\end{aligned}$$

Let's see what we can do with these region operators. First, build an annulus (Figure 14) out of our unit disk, given an inner radius, by subtracting one disk from another:


[$width = 3$]
Fig. 14. *annulus* 0.5

```
annulus :: Frac → Region
annulus inner = udisk \ uscale inner udisk
```

Next, make a region consisting of alternating infinite pie wedges (Figure 15), which is a simplification of Figure 4.

```
radReg :: IntE → Region
radReg n = test ∘ toPolar
  where
    test (r, a) = even ⌊a ∗ fromInt n / π⌋
```

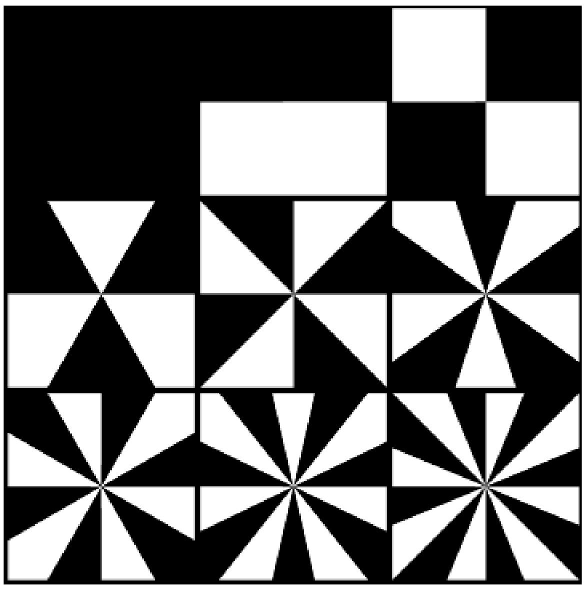
Putting these two together, we get Figure 16.

```
wedgeAnnulus :: Float → Int → Region
wedgeAnnulus inner n = annulus inner ∩ radReg n
```

The *xorR* operator is useful for creating op art. For instance, Figure 17 is made from two copies of *altRings* (Figure 3), shifted in opposite directions and combined with *xorR*.

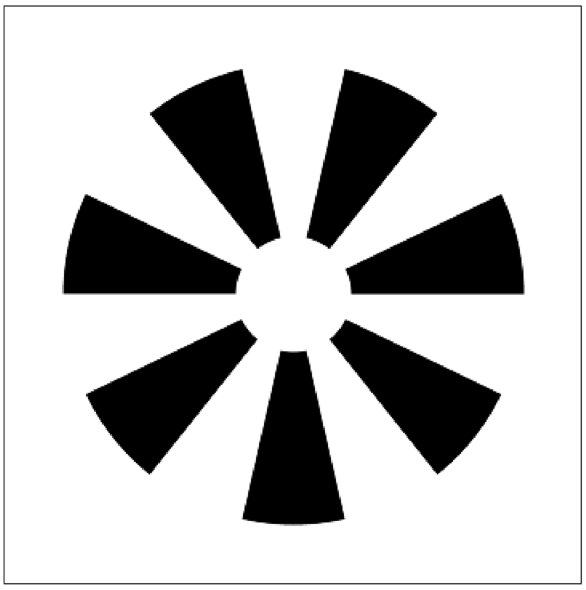
```
shiftXor :: Float → Filter Bool
shiftXor r reg = reg′ r ‘xorR‘ reg′ (−r)
  where
    reg′ d = translate (d, 0) reg
```

Why stop at two copies of *altRings*? For any given n, the following definition



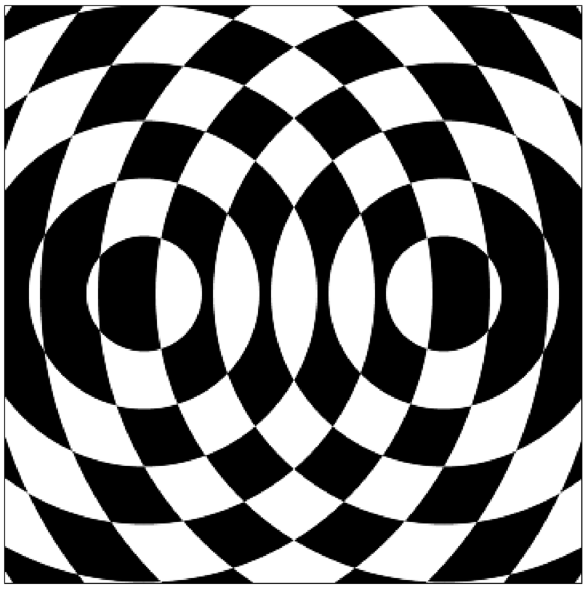
[$duration = 8, width = 1$]

Fig. 15. *radReg* n for $n = 0, \ldots, 8$



[$width = 2.5$]

Fig. 16. *wedgeAnnulus* 0.25 7



$[width = 10]$

Fig. 17. *shiftXor* 2.6 *altRings*

distributes n copies of *altRings* around a circle of radius r and xors them all together (Figure 18).

```
xorgon :: Int → Float → Region → Region
xorgon n r = xorRs (map rf [0 .. n − 1])
  where
    rf i = translate (fromPolar (r, a)) altRings
      where
        a = fromInt i ∗ 2 ∗ π / fromInt n
```

The function *xorRs* does for a list of regions what *xorR* does for two.

```
xorRs :: [Region] → Region
xorRs = foldr xorR emptyR
```

Note also that *polarChecker* (Figure 4) can be redefined very simply applying *xorR* to *altRings* (Figure 3) and *radReg* (Figure 15):

```
polarChecker n = altRings ‘xorR‘ radReg n
```

Similarly, one could use *xorR* and a coordinate-swapping filter to redefine *checker* (Figure 2) in terms of a region with alternating horizontal or vertical slabs,

One use for regions is to crop a color-valued image:

```
type FilterC = Filter Color
crop :: Region → FilterC
crop reg im = cond reg im empty
```

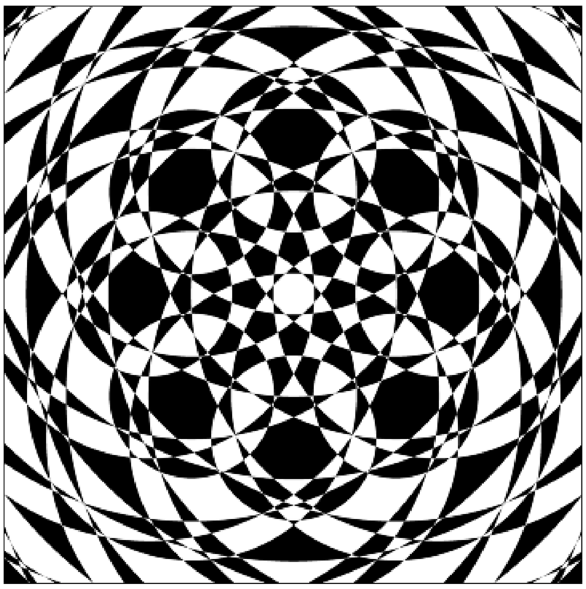


Fig. 18. *xorgon* 8 (7/4) *altRings*

For instance, Figures 19 and 20 come from cropping *ybRings* (Figure 9) with regions produced from *wedgeAnnulus* (Figure 16) and from a swirled version of *wedgeAnnulus*.

8 Tiling

One way to create an infinite image from a finite one (i.e., one transparent outside of a finite area) is *tiling*, which is an infinite repitition with displacement and possibly rotation of a given image. One simple form of tiling is rectangular, in which there is no rotation, and the displacements are all by $(n \cdot w, m \cdot h)$ for arbitrary integers n, m and fixed width and height w, h.

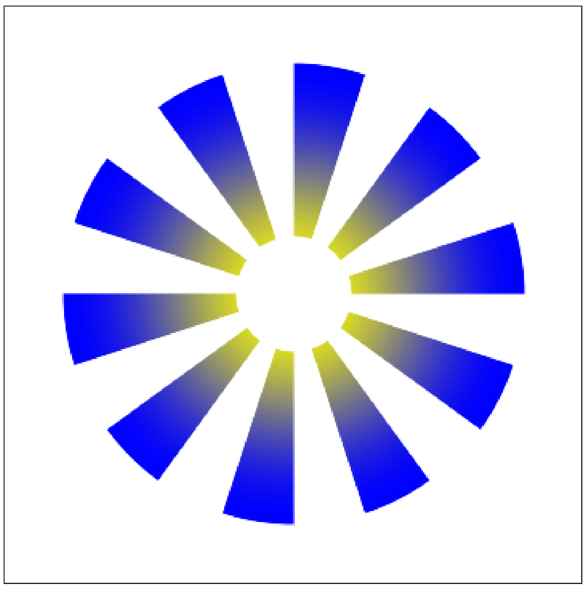
For instance Figure 21 shows a tiling of an image *kids* of size 100 by 117.

The *tile* image-warping function is defined, as usual, in terms of a *Warp*-building function:

$$\begin{aligned}&\mathit{tile} :: \mathit{Vector} \to \mathit{Filter}\ c\\&\mathit{tile}\ \mathit{size} = \mathit{invWarp}\ (\mathit{tileP}\ \mathit{size})\end{aligned}$$

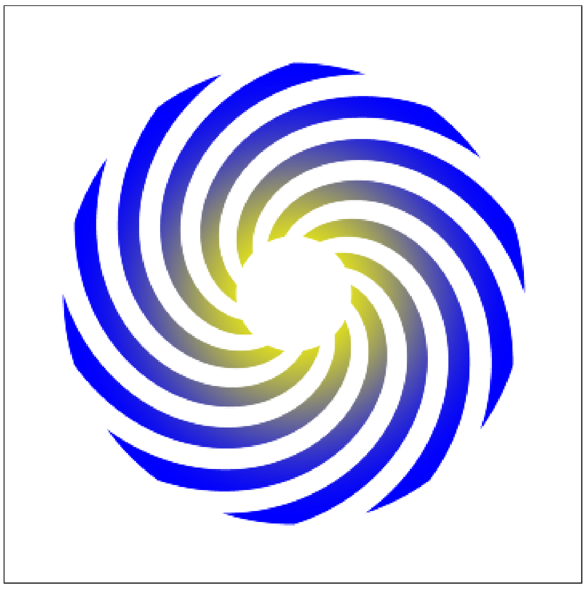
To define *tileP*, first handle the one-dimensional case, using the functions *wrap*, which maps to the range $[0, w)$, and *wrap*′, which maps to $[-w/2, w/2)$.

$$\begin{aligned}&\mathit{wrap}, \mathit{wrap}' :: \mathit{Float} \to \mathit{Float} \to \mathit{Float}\\&\mathit{wrap}\ w\ x = w * \mathit{fracPart}\ (x/w)\\&\mathit{wrap}'\ w\ x = \mathit{wrap}\ w\ (x + w/2) - w/2\\&\mathit{fracPart}\ z = z - \lfloor z \rfloor\end{aligned}$$



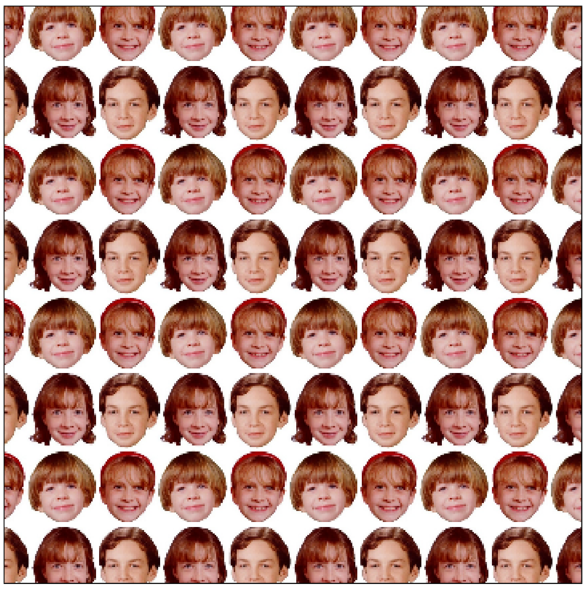
[*width* = 2.5]

Fig. 19. *crop* (*wedgeAnnulus* 0.2 5 10) *ybRings*



[*width* = 2.5]

Fig. 20.
crop (*swirl* 2 (*wedgeAnnulus* 0.25 10)) *ybRings*



$[width = 450]$

Fig. 21. *tile* $(100, 117)$ *kids*

Then we can define *tiling* via two applications of *wrap*′:

$$\begin{aligned}&\mathit{tileP} :: \mathit{Vector} \to \mathit{Warp}\\ &\mathit{tileP}\,(w,h) = \lambda\,(x,y) \to (\mathit{wrap}'\ w\ x,\ \mathit{wrap}'\ h\ y)\end{aligned}$$

Figure 22 shows a tiling of Figure 7.

$$\mathit{tiledBilerp} = \mathit{about}\,(1/2, 1/2)\,(\mathit{tile}\,(1,1))\ \mathit{bilerpBRBW}$$

The higher-order function *about* takes care of tiling about the point $(1/2, 1/2)$ instead of $(0, 0)$.

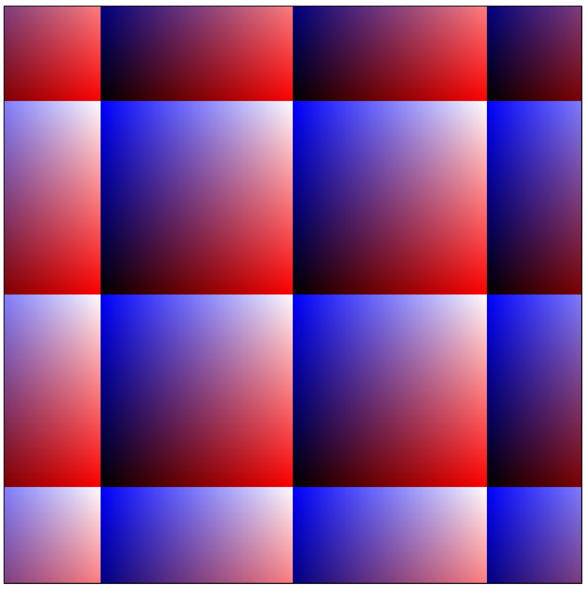
$$\begin{aligned}&\mathit{about} :: \mathit{Point} \to \mathit{HyperFilter}\ c\\ &\mathit{about}\,(x,y)\ \mathit{filt} = \mathit{translate}\,(x,y) \circ \mathit{filt} \circ \mathit{translate}\,(-x,-y)\end{aligned}$$

$$\mathit{type}\ \mathit{HyperFilter}\ c = \mathit{Filter}\ c \to \mathit{Filter}\ c$$

9 Some polar warps

The *swirlP* function (from Section 5 and used to define *swirl*) can be somewhat simplified by considering points in polar rather than rectangular coordinates.[8]

[8] In polar coordinates, a point p is identified by a pair (ρ, θ), where ρ is the distance from the origin and θ is the angle between the positive X axis and the ray emanating fm the origin and passing through p.



$[width = 3]$

Fig. 22. *tiledBilerp*

$$swirlP\ r\ =\ polarWarp\ (\lambda\ (\rho,\theta)\ \rightarrow\ (\rho,\ \theta\ +\ \rho\ *\ (2\pi\ /\ r)))$$

Note that θ changes but ρ does not.

The useful function *polarWarp* is defined very simply:

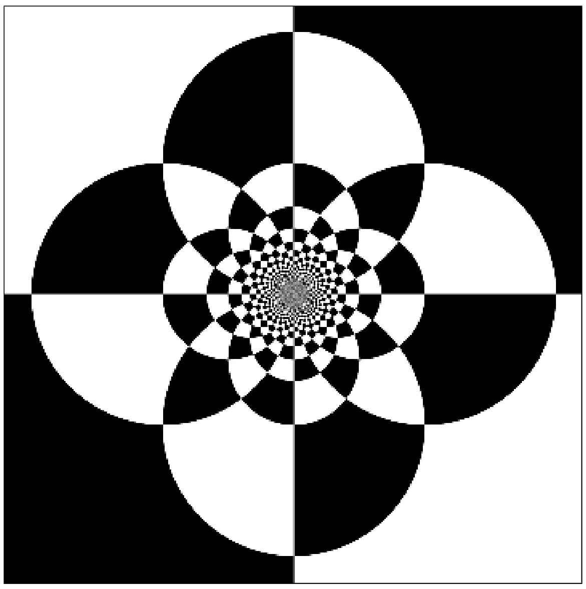
$$\begin{aligned}&polarWarp\ ::\ Warp\ \rightarrow\ Warp\\&polarWarp\ warp\ =\ fromPolar\ \circ\ warp\ \circ\ toPolar\end{aligned}$$

9.1 Turning things inside out

Next, let's consider a polar warp that changes ρ but not θ. Simply multiplying ρ by a constant is equivalent to uniform scaling (*uscale*). However, *inverting* ρ has a striking effect (Figure 23):

$$\begin{aligned}&radInvertP\ ::\ Warp\\&radInvertP\ =\ polarWarp\ (\lambda\ (\rho,\theta)\ \rightarrow\ (1/\rho,\theta))\end{aligned}$$

$$\begin{aligned}&radInvert\ ::\ Filter\ c\\&radInvert\ =\ invWarp\ radInvertP\end{aligned}$$



$[width = 2.2]$

Fig. 23. *radInvert checker*

9.2 Radial ripples

As another radial (ρ) warp, we can multiply ρ by an amount that oscillates around 1 with a given magnitude s, having a given number n of periods as θ varies from 0 to 2π. As usual, define an image-warping version as well:[9]

```
rippleRadP :: Int → Float → Warp
rippleRadP n s = polarWarp $
  λ (ρ, θ) → (ρ ∗ (1 + s ∗ sin (fromInt n ∗ θ)), θ)

rippleRad :: Int → Float → Filter c
rippleRad n s = invWarp (rippleRadP n (−s))
```

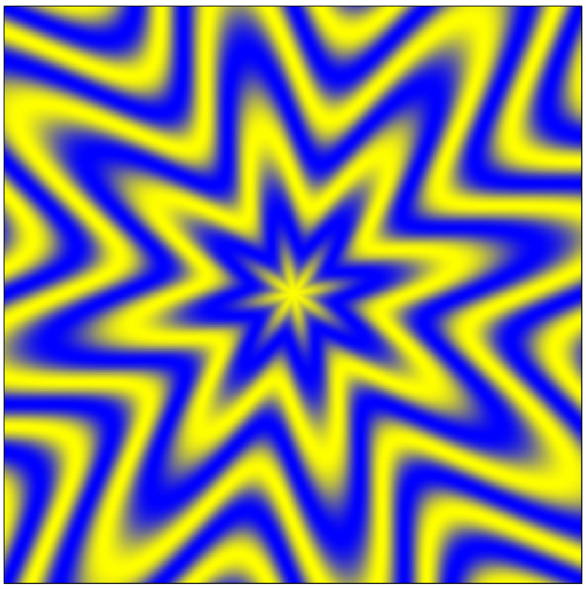
In order to visualize the effect of *rippleRad*, apply it to *ybRings* (Figure 9). As usual, we can use the effect statically or in an animation (Figures 24 and 25).

The examples so far have been infinite in size. We can also make finite ones by cropping against a region. As a convenience, define *cropRad* as a function that crops an image to a disk-shaped region of a given radius:

```
cropRad :: Float → FilterC
cropRad r = crop (uscale r udisk)
```

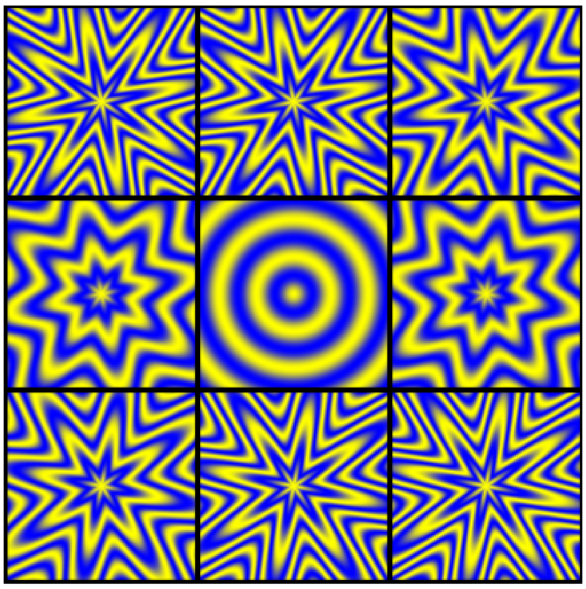
We can crop and then ripple (Figure 26) or ripple and then crop (Figure 27).

[9] The "$" operator is infix, right-associative, low-precedence function application. It often reduces the need for parentheses.



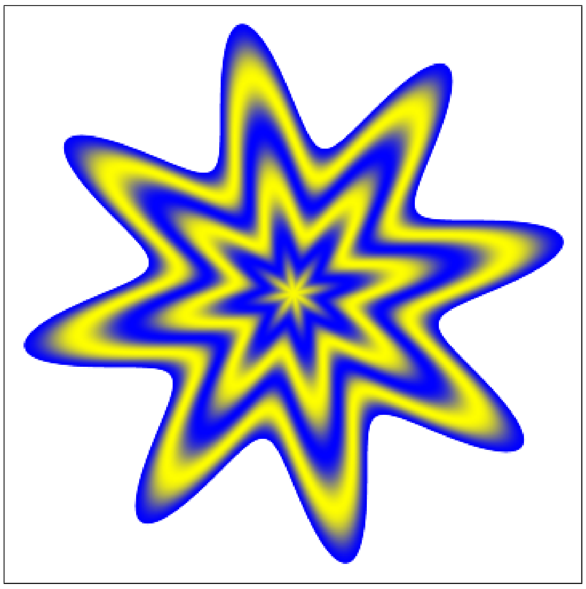
$[width = 10]$

Fig. 24. *rippleRad* 8 0.3 *ybRings*



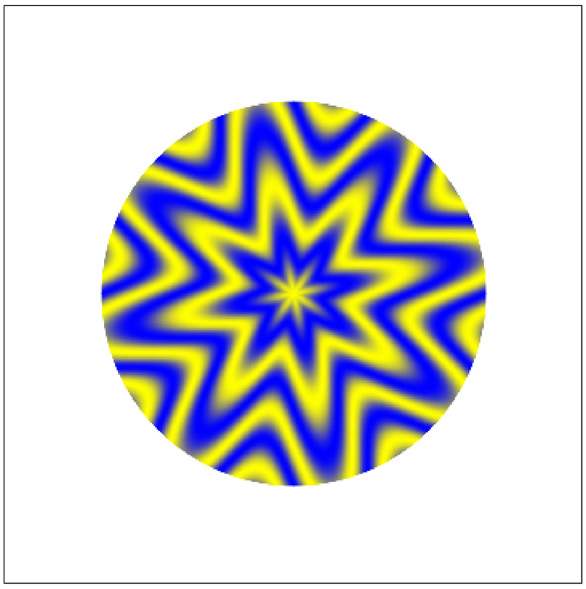
$[duration = \pi, width = 10]$

Fig. 25. $\lambda t \rightarrow$ *rippleRad* 8 $(cos\ t/2)$ *ybRings*



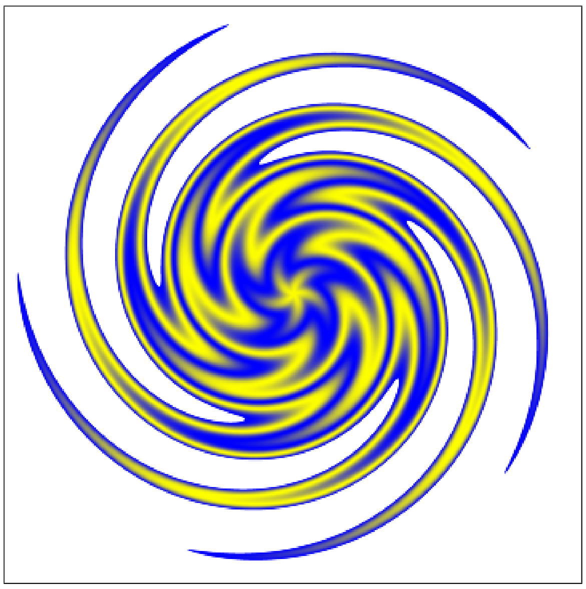
$[width = 15]$

Fig. 26. *rippleRad* 8 0.3 $ *cropRad* 1 $ *ybRings*



$[width = 15]$

Fig. 27. *cropRad* 1 $ *rippleRad* 8 0.3 $ *ybRings*



[*width* = 15]
Fig. 28. *swirl* 8 $ *rippleRad* 5 0.3$ *cropRad* 5 $ *ybRings*

Now let's throw in swirling after rippling. Again the result have quite different looks depending on application order (Figures 29 and 28).[10]

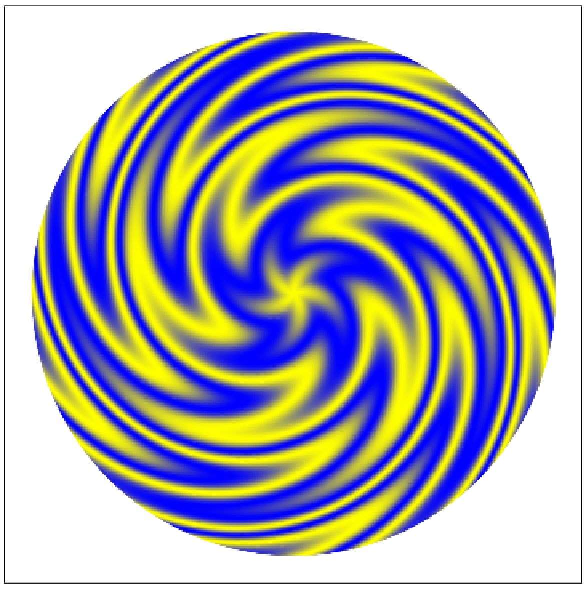
9.3 Radial waves

For yet another radial warp, let's do a scale that ripples based on ρ instead of θ. This time, instead of hardwiring *sin* into the definition, let's take a function parameter:

```
cwaveP :: (Float → Float) → Float → Warp
cwaveP f period = polarWarp warp
  where
    warp (r, a) = (f (r ∗ (2 π / period)), a)
cwave f r = invWarp (cwaveP f r)
```

The *tan* and *sin* functions yield quite different results. For aesthetics, we also apply a rotation and circular crop to the *sin* example (Figures 30 and 31).

[10] If we were to swap the order of *swirl* and *cropRad* in Figure 28, the results would be disappointing. Why?



$[width = 10]$
Fig. 29. *swirl* 8 \$ *cropRad* 5\$ *rippleRad* 5 0.3 \$ *ybRings*

9.4 The washing machine

Let's imagine that we're watching a load of colorful clothes in an agitator-style washing machine. As the agitator turns in the center, it takes a while for the twisting motion to radiate outward. The result might look something like Figure 32.

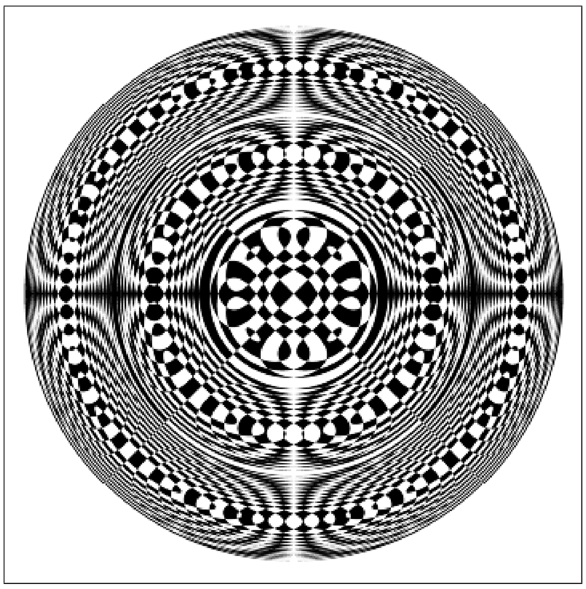
The heart of this example is the warp-constructing function *wiggleRotateP*, which takes a number *cycles* of oscillation cycles per unit from the origin, and a maximum rotation angle:

```
wiggleRotateP :: Float → Float → Time → Warp
wiggleRotateP cycles θ_max t = polarWarp warp
  where
    warp (r, a) = (r, a + θ_max * sin (t + dt))
      where
        dt = 2 π * cycles * (r − 1/2)

wiggleRotate :: Float → Float → Time → Filter c
wiggleRotate cycles θ_max t = invWarp (wiggleRotateP cycles θ_max t)
```

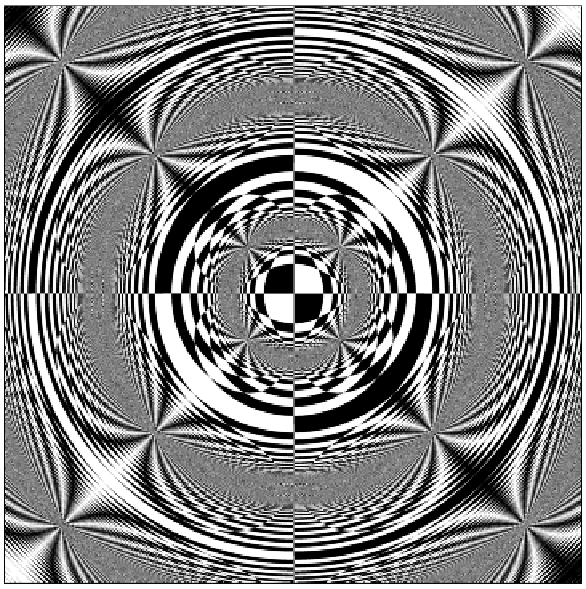
To form the washer image in Figure 32, take a smoothly colored initial image *tiledBilerp*, warp it with *wiggleRotate*, and crop it with a disk:

```
washer :: Float → Float → ImageC → Time → ImageC
washer cycles θ_max im t = cropRad 1 $ wiggleRotate cycles θ_max t $ im
```



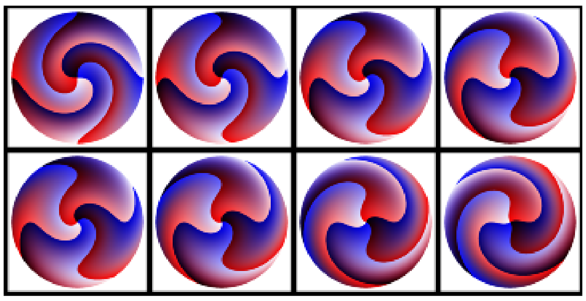
[*width* = 45]

Fig. 30.
uscale 21 *udisk* ∩
rotate (π/4) (*cwave sin* 14 *checker*)



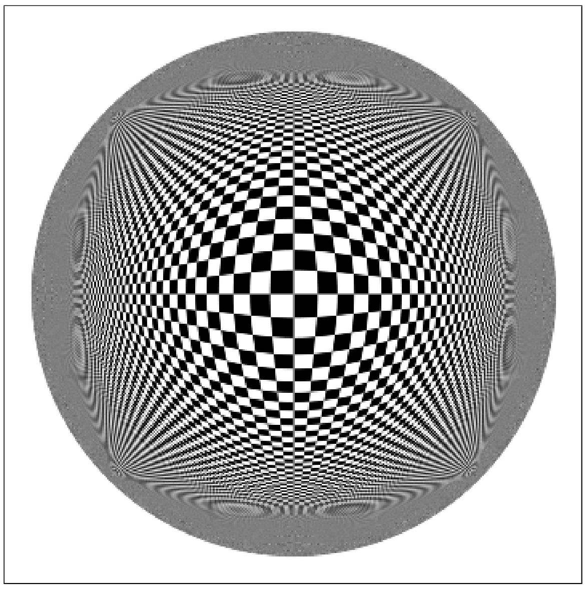
[*width* = 22]

Fig. 31. *cwave tan* 10 *checker*



[$duration = \pi$, $width = 2.1$]

Fig. 32. *washer* (1/2) (π/2) 1 *tiledBilerp*



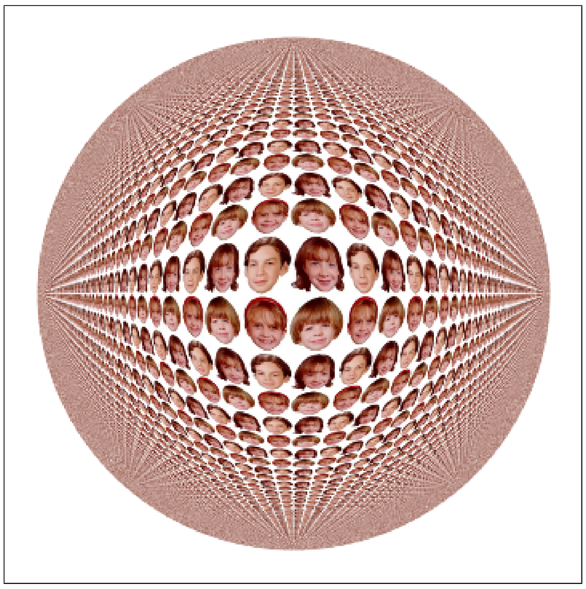
[$width = 22$]

Fig. 33. *circleLimit* 10 (*blackWhiteIm checker*)

9.5 Circle limits

Figure 33 shows the result of squeezing our infinite *checker* image into a finite disk. Note that the spatial warp used is essentially one-dimensional. It just moves a point closer or further from the origin, based only on its given distance.

```
circleLimit :: Float → FilterC
circleLimit radius im = cropRad radius (im ∘ polarWarp warp)
  where
    warp (r, a) = (radius ∗ r/(radius − r), a)
```



[$width = 450$]

Fig. 34. *circleLimit* 200 (*tile* (102, 120) *kids*)

Tiling and *circleLimit* go well together, as illustrated in Figure 34. Iterating this pair of operations results in images like Figure 35.

$$\begin{aligned} &\mathit{tileLimit} :: \mathit{Float} \to \mathit{Int} \to \mathit{FilterC} \\ &\mathit{tileLimit}\ r\ n = (\mathit{tile}\ (2.1 * r, 2.1 * r) \circ \mathit{circleLimit}\ r)^n \end{aligned}$$

where

$$\begin{aligned} f^0\, x &= x \\ f^n x &= f^{n-1}\,(f\ x), \text{ for } n > 0 \end{aligned}$$

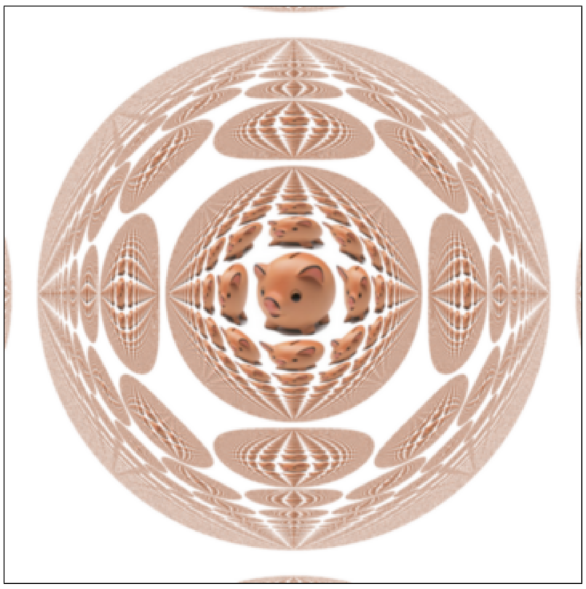
10 Strange hybrids

Regions are useful for cropping images, as in *cropRad* above, but also for pointwise selection, using *cond* (Section 4). For instance, *cond xPos im im′* looks like *im* in its right half-space and like *im* in its left half-space.

To create more interesting images, warp the basic *xPos* region before applying selection. For convenience in constructing examples, let's define a function to select between a girl and her cat, based on a given time-varying region:

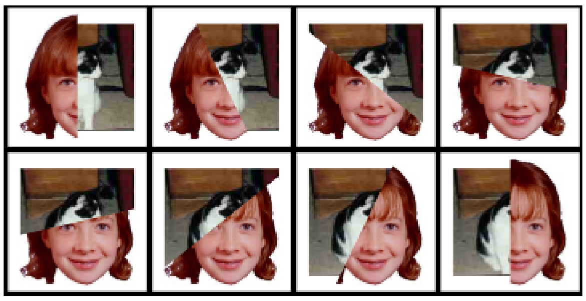
$$\begin{aligned} &\mathit{hybrid} :: (\mathit{Time} \to \mathit{Region}) \to (\mathit{Time} \to \mathit{ImageC}) \\ &\mathit{hybrid}\, f\ t = \mathit{cond}\ (f\ t)\ \mathit{fraidy}\ \mathit{becky} \end{aligned}$$

Appendix A describes the embedding of photographic images in our image model. Figures 36 through 38 show some animations based on a few time-varying regions:



$[width = 450]$

Fig. 35. *tileLimit* 200 2 (*tile* (105, 105) *piggy*)



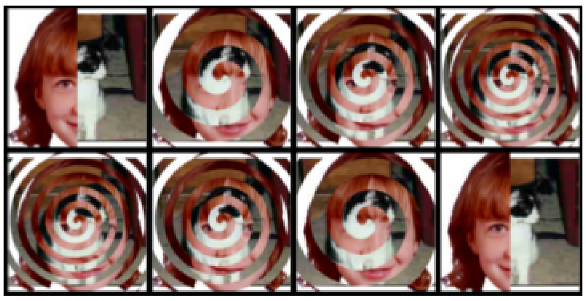
$[duration = \pi, width = 120]$

Fig. 36. *hybrid turningXPos*

$$\begin{aligned} turningXPos\ t &= rotate\ t\ xPos \\ swirlingXPost &= swirl\ (10\ /\ sin\ t)\ xPos \\ roamingDisk\ t &= uscale\ 30\ (translate\ (cos\ t,\ sin\ (2 * t))\ udisk) \end{aligned}$$

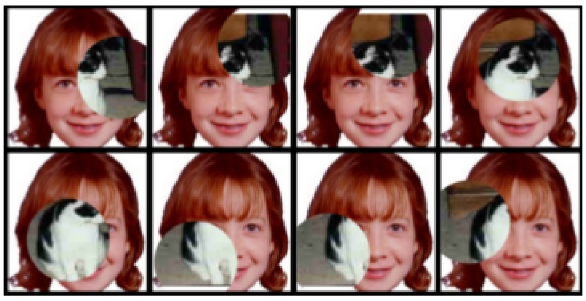
The *cond* function produces hard edges between the images being combined. For a gentler blending, we can shift gradually from one image to the other, using *lerpI* (Section 4), if we can construct a space-varying fraction for *lerpI*'s first argument. As a first step, the following continuous function has value 0 for $x < -1/2$, 1 for $x > 1/2$, and grows linearly in-between:

$$wipe_1\ ::\ Float\ \rightarrow\ Frac$$



$[duration = \pi, width = 120]$

Fig. 37. *hybrid swirlingXPos*



$[duration = \pi, width = 120]$

Fig. 38. *hybrid roamingDisk*

$$wipe_1\ x\ =\ clamp\ 0\ 1\ (x\ +\ 1/2)$$

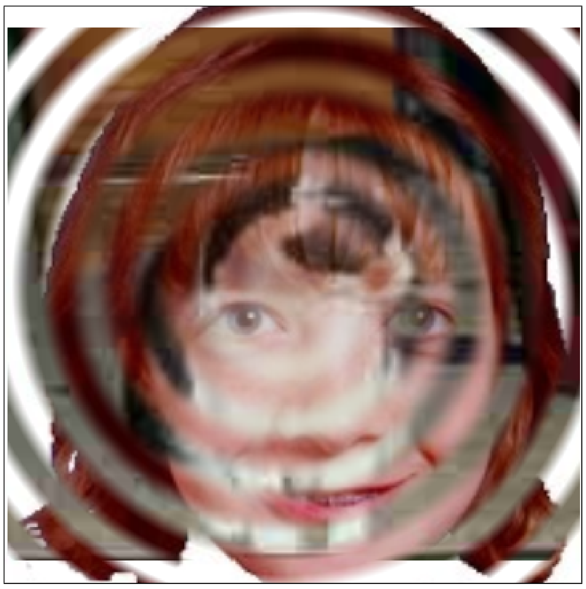
$$clamp\ ::\ Float\ \rightarrow\ Float\ \rightarrow\ Float\ \rightarrow\ Float$$
$$clamp\ lo\ hi\ z\ =\ max\ lo\ (min\ z\ hi)$$

To define a continuous counterpart to *xPos*, we extend $wipe_1$ to 2D and scale by a given transition width w. Figure 39 shows use of $wipe_2$ with *lerpI*:

$$wipe_2\ ::\ Float\ \rightarrow\ Image\ Frac$$
$$wipe_2\ w\ (x, y)\ =\ wipe_1\ (x/w)$$

11 Related Work

Peter Henderson began the game of functional *geometry* for image synthesis (Henderson, 1982). Since then there have been several other such libraries (Lucas & Zilles, 1987; Zilles *et al.*, 1988; Bartlett, 1991; Arya, 1994; Finne & Jones, 1995; Elliott & Hudak, 1997). Many or all of these libraries are based on a spatially continuous model, but unlike Pan, none has addressed the general notion of images. Similarly for the various "vector-based" 2D APIs and file formats.



[*width* = 120]
Fig. 39. *lerpI* (*swirl* 10 (*wipe*$_2$ 75)) *becky fraidy*

Gerard Holzmann developed a system called "Pico", which consisted of an editor, a simple language for image transformations, and a machine-code generator for fast display. His delightful book shows many examples, using photos of Bell Labs employees (Holzmann, 1988). Pico's model of images was the discrete rectangular array of bytes, which could be interpreted as grey-scale values or other scalar fields. The host language appears to have been very primitive, with essentially no abstraction mechanisms.

John Maeda's "Design by Numbers" (DBN) is another language aimed at simplifying image synthesis, sharing with Pan the goals of simplicity and encouragement of creative exploration (Maeda, 1999). In contrast, the DBN language is squarely in the imperative style (*doing* rather than *being*). Its programs are lists of commands for outputting dots or line segments and changing internal state, with an image emerging as the cumulative result. Like Pico, DBN presents a discrete notion of space, partitioned into a finite array of square pixels.

The Haskell "region server" (Hudak & Jones, 1994) used characteristic functions to represent regions, in essentially the same formulation as Pan (Section 7). Those regions were not used for visualization, nor were they generalized to range types other than Boolean. Paul Hudak also used regions for graphics (Hudak, 2000). There an algebraic data-type represents regions, but an interpretation (semantics) is given by translating to a function from 2D space to Booleans.

In his work on evolution for computer graphics, Karl Sims represented images as Lisp expressions over variables with names x, y, and t (adding z for solid textures) (Sims, 1991). He did not exploit Lisp's support for higher-order functional programming for composing image functions.

12 Conclusions

For the purpose of image synthesis, imperative programming languages (including most object-oriented ones) are unpleasantly "low-level" in the sense of Alan Perlis: "A programming language is low level when its programs require attention to the irrelevant."

This paper presents a simple model and high-level *functional* programming language for images, as functions from continuous 2D space to colors, and then tests the expressiveness of this model by means of several examples. These examples represent just a hint at what can be done, and are far from exhaustive, or even necessarily representative. I hope that readers are inspired to apply their own creativity to generate images and animations that look very different from the examples in this paper.

13 Acknowledgements

Sigbjørn Finne and Oege de Moor collaborated on the implementation and provided fruitful discussions. Brian Guenter suggested the project of an image manipulation language and optimizing compiler and has provided helpful expertise. Koen Classen provided helpful comments on an earlier draft of this paper.

References

Arya, K. (1994). A functional animation starter-kit. *Journal of functional programming*, **4**(1), 1–18.

Bartlett, J. F. 1991 (May). *Don't fidget with widgets, draw!* Tech. rept. 6. DEC Western Digital Laboratory, 250 University Avenue, Palo Alto, California 94301, US.

Elliott, C. (1999). An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE transactions on software engineering*, **25**(3), 291–308. Special Section: Domain-Specific Languages (DSL). `http://research.microsoft.com/~conal/papers/tse-modeled-animation`.

Elliott, C., & Hudak, P. 1997 (9–11 June). Functional reactive animation. *Pages 263–273 of: Proceedings of the 1997 ACM SIGPLAN international conference on functional programming.* `http://research.microsoft.com/~conal/papers/icfp97.ps`.

Elliott, C., Finne, S., & de Moor, O. (2001). *Compiling embedded languages.* To appear in the *Journal of Functional Programming.* See `http://research.microsoft.com/~conal/papers/saig00` for an earlier version.

Finne, S., & Jones, S. Peyton. 1995 (July). Pictures: A simple structured graphics model. *Glasgow functional programming workshop.*

Henderson, P. (1982). Functional geometry. *Pages 179–187 of: ACM symposium on LISP and functional programming.*

Holzmann, G. J. (1988). *Beyond photography — the digital darkroom.* Englewood Cliffs, New Jersey: Prentice-Hall. (Out of print).

Hudak, P. (1998). Modular domain specific languages and tools. *Pages 134–142 of:* Devanbu, P., & Poulin, J. (eds), *Proceedings: Fifth international conference on software reuse.* IEEE Computer Society Press.

Hudak, P. (2000). *The Haskell school of expression – learning functional programming through multimedia.* New York: Cambridge University Press.

Hudak, P., & Jones, M. P. (1994). *Haskell vs. Ada vs. C++ vs Awk vs . . . an experiment in software prototyping productivity.* Tech. rept. Yale.

Jones, S.L. Peyton, Hughes, R.J.M., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M.P., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., & Wadler, P.L. 1999 (Feb.). *Haskell 98: A non-strict, purely functional language.* `http://haskell.org/definition`.

Lucas, P., & Zilles, S. N. 1987 (July 8). *Graphics in an applicative context.* Tech. rept. IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099.

Maeda, J. (1999). *Design by numbers.* MIT Press. Foreword by Paola Antonelli. `http://www.maedastudio.com/dbn`.

Mandelbrot, B. B. (1977). *The fractal geometry of nature.* New York: W.H. Freeman & Company.

Sims, K. (1991). Artificial evolution for computer graphics. *ACM computer graphics*, **25**(4), 319–328. SIGGRAPH '91 Proceedings.

Smith, A. R. 1995 (July). *Image compositing fundamentals.* Tech. rept. Technical Memo #4. Microsoft. `http://www.alvyray.com/Memos`.

Zilles, S.N., Lucas, P., Linden, T.M., Lotspiech, J.B., & Harbury, A.R. 1988 (December 5–9). The Escher document imaging model. *Pages 159–168 of: Proceedings of the ACM conference on document processing systems (Santa Fe, New Mexico).*

A Bitmaps

Image importation must make up two differences between our "image" notion and the various "bitmap" formats that can be imported.[11] Pan images have infinite domain and are continuous, while bitmaps are finite and discrete arrays, which we represent as dimensions and a subscripting function:

$$data\ Array_2\ c\ =\ Array_2\ Int\ Int\ ((Int,\ Int)\ \rightarrow\ c)$$

That is, $Array_2\ n\ m\ f$ represents an array of n columns and m rows, and the valid arguments (indices) of f are in the range $\{0, \ldots, n-1\} \times \{0, \ldots, m-1\}$.

Rather than creating and storing an actual array of colors (quadruples of floating point numbers), conversion from the file representation (typically 1, 8, 16, or 24 bits) is done on-the-fly during "subscripting". The details depend on the particular format. This flexibility is exactly why we chose to use subscripting functions rather than a more concrete representation.

The heart of the conversion from bitmaps to images is captured in the *reconstruct* function. Sample points outside of the array's rectangular region are mapped to the invisible color. Inner points generally do not map to one of the discrete set of pixel locations, so some kind of filtering is needed. For simplicity with reasonably

[11] Somewhat misleadingly, the term "bitmap" is often used to refer not only to monochrome (1-bit) formats, but to color ones as well.

good results, Pan uses bilinear interpolation (*bilerp*, from Section 3), to performs a weighted average of the four nearest neighbors. Given any sample point p, find the four pixels nearest to p and bilerp the four colors, using the position of p relative to the four pixels. (Note that dx and dy are fractions.)

```
bilerpArray2 :: ((Int, Int) → Color) → ImageC
bilerpArray2 sub (x, y) =
  let
    i = ⌊x⌋; wx = x − fromInt i
    j = ⌊y⌋; wy = y − fromInt j
  in
  bilerp (sub (i, j    )) (sub (i + 1, j    ))
         (sub (i, j + 1)) (sub (i + 1, j + 1))
         (wx, wy)
```

Finally, define reconstruction of a bitmap into an infinite extent image. The reconstructed bitmap will be given by *bilerpArray*$_2$ inside the array's spatial region, and empty (transparent) outside. For convenience, the region is centered at the origin:

```
reconstruct :: Array2 Color → ImageC
reconstruct (Array2 w h sub) =
  move (− fromInt w / 2, − fromInt h / 2)
       (crop (inBounds w h) (bilerpArray2 sub))
```

The function *inBounds* takes the array bounds (width w and height h), and checks whether a point falls within the given array bounds:

```
inBounds :: Int → Int → Region
inBounds w h (x, y) = 0 ≤ x ∧ x ≤ fromInt (w − 1) ∧
                      0 ≤ y ∧ y ≤ fromInt (h − 1)
```