The Fun of Programming

Edited by

Jeremy Gibbons and Oege de Moor

Contents

7 Functional images

Conal Elliott

7.1 Introduction

Functional programming offers its practitioners 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 chapter 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 specialised 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 [34], is very valuable, in that it allows one to concentrate on the underlying model and rely on a library to take care of presentation. This focus makes it easier to describe images (for example, via a set of curve control points or a text string), while narrowing the range of describable images. 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. Fortunately, as illustrated in this chapter, the *general* notion of images may be modelled directly and effectively *as functions* from a 2-dimensional domain to colours. 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.

This chapter explores the very simple notion of images as functions in a Haskell library — a 'domain-specific embedded language' (DSEL) [60] — called

Pan. It presents the types and operations that make up Pan, and illustrates their use through a collection of examples. Some of the examples are synthesised from mathematical descriptions, while others are image-transforming 'filters' that can be applied to photographs or synthetic images. For more examples, including colour and animations, see the example gallery at this book's supporting web site; the implementation is freely available for downloading from there.

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 colour-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 colour 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, laziness has not been necessary, so translation to a strict functional language should be straightforward and satisfactory.

For efficiency, Pan is implemented as a compiler [35].[1] 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 optimising compiler.

7.2 What is an image?

Pan's model of images is simply functions from infinite, continuous 2D space to colours 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]

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

where *Point* is the type of Cartesian coordinates:

$$\textbf{type}\ \mathit{Point} = (\mathit{Float}, \mathit{Float})$$

[1] At the time of writing, running the Pan compiler requires having the Microsoft C++ compiler.

[2] As described elsewhere [35], the implementation really uses 'expression types' with names like *FloatE* instead of *Float*, in order to optimise 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 chapter could, however, be used directly as a valid but less efficient implementation. For conciseness, this chapter uses some standard mathematical notation for functions like absolute value, floor, and square root.

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

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

It can also be useful to generalise the domain of images, from points in 2D space to other types (such as 3D space or points with integer coordinates). 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 7.1 shows an infinitely tall vertical strip of unit width, *vstrip*, as defined below.[3]

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

For a slightly more complex example, consider the checkered region shown in Figure 7.2. 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 $\lfloor x \rfloor + \lfloor y \rfloor$ changes.

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

Images need not have straight edges and right angles. Figure 7.3 shows a collection of concentric black and white rings. The definition is similar to *checker*, but uses the distance from the origin to a given point, as computed by *distO*.

$$\mathit{altRings}\ p = \mathit{even}\ \lfloor \mathit{distO}\ p \rfloor$$

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

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

It is often more convenient to define images using polar coordinates (ρ, θ) (where ρ is the distance from the origin, and θ is the angle between the positive X axis and the ray emanating from the origin and passing through p), rather than rectangular coordinates (x, y).

$$\textbf{type}\ \mathit{PolarPoint} = (\mathit{Float}, \mathit{Float})$$

[3] Each figure shows an origin-centred finite window onto an infinite image and is annotated with the width of the window in logical coordinates. For instance, Figure 7.1 shows the window $[-7/2, 7/2] \times [-7/2, 7/2]$ onto the infinite *vstrip* image.

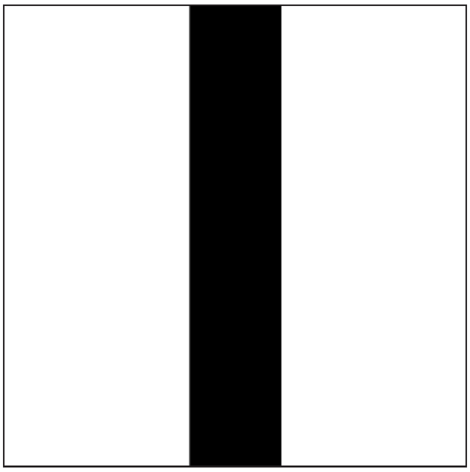


Figure 7.1: *vstrip*

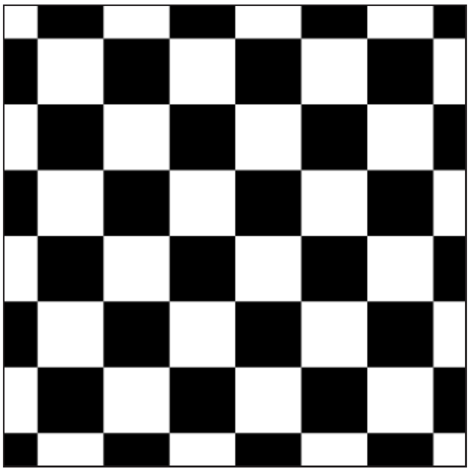


Figure 7.2: *checker*

The following definitions are helpful for converting between polar and rectangular coordinates.

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

Figure 7.4 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.[4] (We will see a simpler definition of *polarChecker* in Section 7.7.)

$$\begin{array}{l}
\mathit{polarChecker} :: \mathit{Int} \rightarrow \mathit{Region} \\
\mathit{polarChecker}\ n = \mathit{checker} \cdot \mathit{sc} \cdot \mathit{toPolar} \\
\quad \textbf{where} \\
\quad\quad \mathit{sc}\ (\rho, \theta) = (\rho, \theta \times \mathit{fromInt}\ n / \pi)
\end{array}$$

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

$$\textbf{type}\ \mathit{Frac} = \mathit{Float}$$

Figure 7.5 shows a wavy grey-scale image that shifts smoothly between white (zero) and black (one) in concentric rings.

[4] *fromInt* turns an integer into some other type, here *Float*.

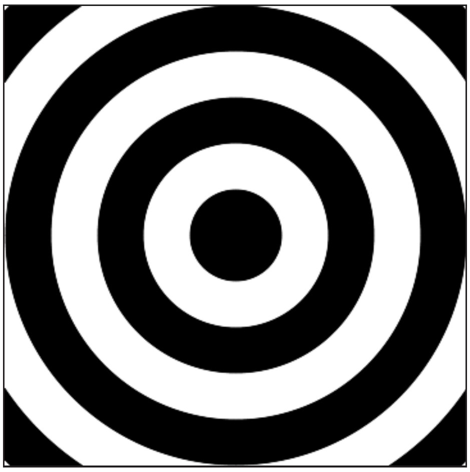


Figure 7.3: *altRings*

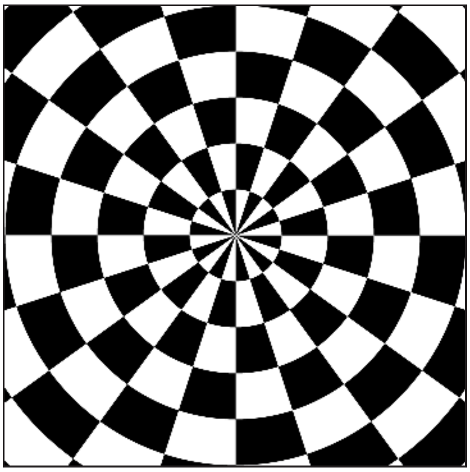


Figure 7.4: *polarChecker* 10

$$
\begin{aligned}
&wavDist :: Image\ Frac\\
&wavDist\ p\ =\ (1\ +\ cos\ (\pi\ \times\ distO\ p))\ /\ 2
\end{aligned}
$$

7.3 Colours

Pan colours 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'):

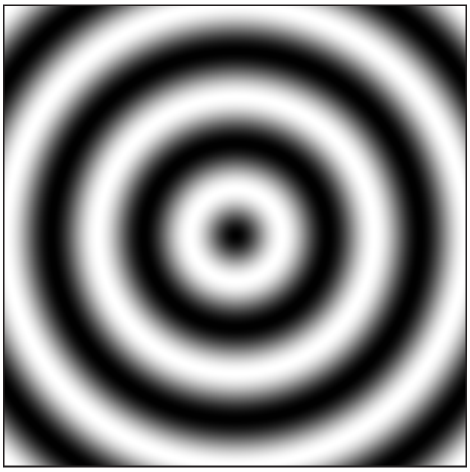
$$\textbf{type}\ Colour = (Frac, Frac, Frac, Frac) \quad \text{— BGRA}$$

The blue, green, and red components will have alpha multiplied in already, and so must be less than or equal to alpha (according to the convention of 'pre-multiplied alpha' [121]). Given this constraint, there is exactly one fully transparent colour:

$$invisible\ =\ (0,0,0,0)$$

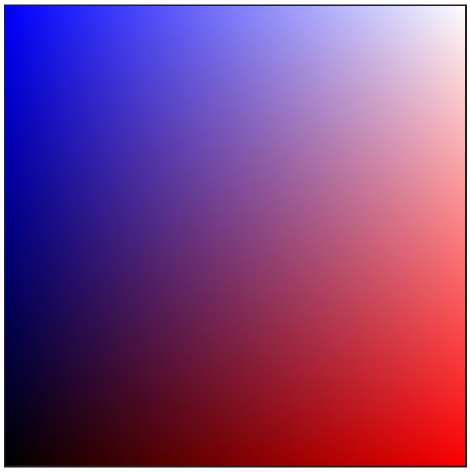
We are now in a position to define some familiar (completely opaque) colours:

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



width = 10

Figure 7.5: *wavDist*



width = 1

Figure 7.6: *bilerpC black red blue white*

It is often useful to interpolate ('lerp') between colours, to create a smooth transition through space or time. This is the purpose of $\mathit{lerpC}\ w\ c_1\ c_2$. The first parameter w is a fraction, indicating the relative weight of the colour c_1. The weight assigned to the second colour c_2 is $1 - w$:

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

Exercise 7.1 Use *lerpC* to define functions that lighten, darken, and fade (toward invisibility) colours by fractional amounts. □

Exercise 7.2 Extend *lerpC* to a function *bilerpC* that interpolates among four colours in two dimensions. (Hint: make three applications of *lerpC*.) See Figure 7.6, which is centred at $(1/2, 1/2)$ rather than the origin. □

An operation similar to *lerpC* is colour overlay, which will be used in the next section to define image overlay. The result is a blend of the two colours, depending on the opacity of the top (first) colour. A full discussion of this definition can be found in [121]:

$$
\begin{aligned}
&(b_1, g_1, r_1, a_1)\ \text{`}\mathit{overC}\text{`}\ (b_2, g_2, r_2, a_2) = (h\ b_1\ b_2, h\ g_1\ g_2, h\ r_1\ r_2, h\ a_1\ a_2)\\
&\quad \textbf{where}\\
&\qquad h\ x_1\ x_2 = x_1 + (1 - a_1) \times x_2
\end{aligned}
$$

Not surprisingly, colour-valued images are of particular interest, so let us use a convenient abbreviation:

$$\textbf{type}\ \mathit{ImageC} = \mathit{Image\ Colour}$$

7.4 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 *overC*:

$$\begin{aligned}&\mathit{over} :: \mathit{ImageC} \to \mathit{ImageC} \to \mathit{ImageC}\\&(\mathit{top}\ \text{`}\mathit{over}\text{`}\ \mathit{bot})\ p = \mathit{top}\ p\ \text{`}\mathit{overC}\text{`}\ \mathit{bot}\ p\end{aligned}$$

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

$$\begin{aligned}&\mathit{lift}_1 :: (\alpha \to \beta) \to (p \to \alpha) \to (p \to \beta)\\&\mathit{lift}_2 :: (\alpha \to \beta \to \gamma) \to (p \to \alpha) \to (p \to \beta) \to (p \to \gamma)\\&\mathit{lift}_3 :: (\alpha \to \beta \to \gamma \to \delta) \to (p \to \alpha) \to (p \to \beta) \to (p \to \gamma) \to (p \to \delta)\end{aligned}$$

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

Then $\mathit{over} = \mathit{lift}_2\ \mathit{overC}$. Other examples of pointwise lifting include selection (*cond*) and interpolation (*lerpI*) between two images:[6]

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

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

Nullary lifting is already provided by Haskell's *const* function:

$$\begin{aligned}&\mathit{const} :: \alpha \to (p \to \alpha)\\&\mathit{const}\ a\ p = a\end{aligned}$$

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

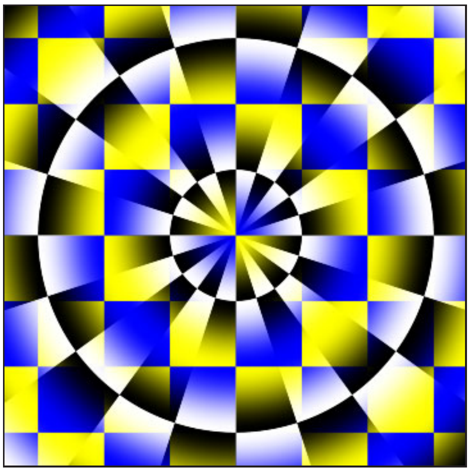
$$\begin{aligned}&\mathit{empty} = \mathit{const\ invisible}\\&\mathit{whiteI} = \mathit{const\ white}\end{aligned}$$

[5]For intuition, think of p as *Point*, so that $p \to \alpha = \mathit{Image}\ \alpha$ and similarly for β, γ, δ.

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

width = 10

Figure 7.7: *ybRings*



width = 7

Figure 7.8: Exercise 7.4

$$\begin{aligned} \mathit{blackI} &= \mathit{const\ black} \\ \mathit{redI} &= \mathit{const\ red} \\ &\ldots \end{aligned}$$

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

The image defined below (shown in Figure 7.7) interpolates between blue and yellow, and will be useful in later examples.

$$\mathit{ybRings} = \mathit{lerpI\ wavDist\ blueI\ yellowI}$$

Exercise 7.3 Define functions *bwIm* and *byIm* that map regions into black-and-white and blue-and-yellow images, respectively. □

Exercise 7.4 Express Figure 7.8 as image interpolation (*lerpI*) of the examples in Figures 7.5, 7.2, and 7.4. Since *lerpI* works on colour images, first colour the rectangular and polar checkerboards with *bwIm* and *byIm*, respectively. □

Exercise 7.5 Define 'colour-lifting' functionals $\mathit{clift}_1, \ldots,$ analogous to $\mathit{lift}_1, \ldots,$ and use clift_2 to simplify the definitions of *lerpC* and *overC* from Section 7.3. □

7.5 Spatial transforms

In computer graphics, spatial transforms are commonly 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 transforms are often thought of as *being* matrices. A simpler and more general point of view, however, is that transforms are simply space-to-space functions:

$$\mathbf{type}\ \mathit{Transform} = \mathit{Point} \rightarrow \mathit{Point}$$

It is then easy to define the familiar affine transforms:

$$\mathbf{type}\ \mathit{Vector} = (\mathit{Float}, \mathit{Float})$$

$$\begin{aligned}
&\mathit{translateP} :: \mathit{Vector} \rightarrow \mathit{Transform}\\
&\mathit{translateP}\ (dx, dy)\ (x, y) = (x + dx, y + dy)
\end{aligned}$$

$$\begin{aligned}
&\mathit{scaleP} :: \mathit{Vector} \rightarrow \mathit{Transform}\\
&\mathit{scaleP}\ (sx, sy)\ (x, y) = (sx \times x, sy \times y)
\end{aligned}$$

$$\begin{aligned}
&\mathit{uscaleP} :: \mathit{Float} \rightarrow \mathit{Transform} \quad \text{— uniform}\\
&\mathit{uscaleP}\ s = \mathit{scaleP}\ (s, s)
\end{aligned}$$

$$\begin{aligned}
&\mathit{rotateP} :: \mathit{Float} \rightarrow \mathit{Transform}\\
&\mathit{rotateP}\ \theta\ (x, y) = (x \times \mathit{cos}\ \theta - y \times \mathit{sin}\ \theta, y \times \mathit{cos}\ \theta + x \times \mathit{sin}\ \theta)
\end{aligned}$$

By definition, transforms map points to points. Can we 'apply' them, in some sense, to map images into transformed images?

$$\mathit{applyTrans} :: \mathit{Transform} \rightarrow \mathit{Image}\ \alpha \rightarrow \mathit{Image}\ \alpha$$

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

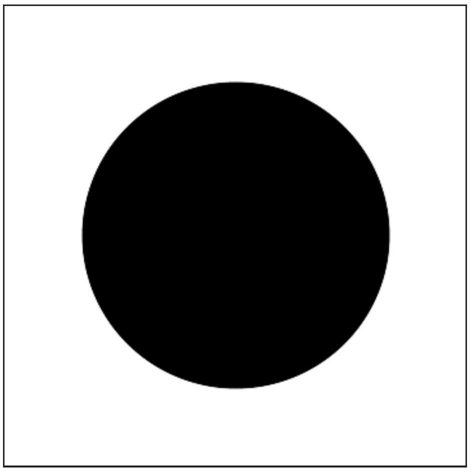
$$\mathit{applyTrans}\ \mathit{xf}\ \mathit{im} \stackrel{?}{=} \mathit{im} \cdot \mathit{xf} \quad \text{— } \mathbf{wrong}$$

Figures 7.9 and 7.10 show a unit disk *udisk* and the result of $\mathit{udisk} \cdot \mathit{uscaleP}\ 2$.

$$\begin{aligned}
&\mathit{udisk} :: \mathit{Region}\\
&\mathit{udisk}\ p = \mathit{distO}\ p < 1
\end{aligned}$$

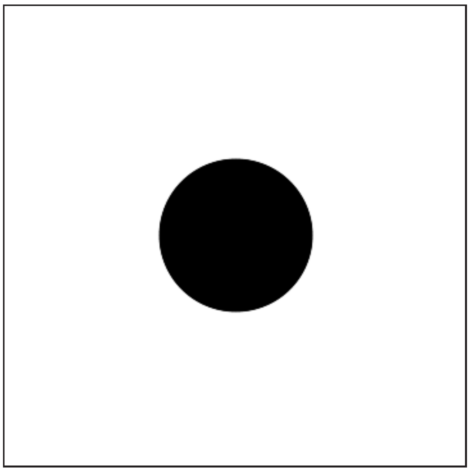
Notice that the *uscaleP*-composed *udisk* is *half* rather than twice the size of *udisk*. (Similarly, $\mathit{udisk} \cdot \mathit{translateP}\ (1, 0)$ moves *udisk* to the *left* rather than right.) The reason is that *uscaleP* 2 maps input points to be twice as far from the origin, so points have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

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



width = 3

Figure 7.9: *udisk*



width = 3

Figure 7.10: *udisk* · *uscaleP* 2

$$\mathit{applyTrans}\;\mathit{xf}\;\mathit{im} \;=\; \mathit{im} \cdot \mathit{xf}^{-1}$$

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 transforms themselves. For these reasons, we simply construct the transforms in inverted form, and do not use *applyTrans*.[7]

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

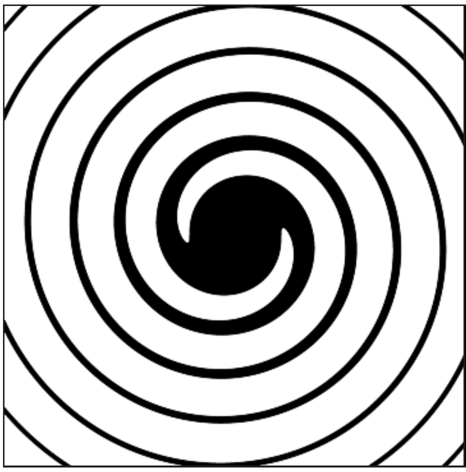
$$\textbf{type}\ \mathit{Filter}\ \alpha = \mathit{Image}\ \alpha \to \mathit{Image}\ \alpha$$

$$\begin{array}{lll} \mathit{translate}, \mathit{scale} & :: \mathit{Vector} & \to \mathit{Filter}\ \alpha \\ \mathit{uscale}, \mathit{rotate} & :: \mathit{Float} & \to \mathit{Filter}\ \alpha \end{array}$$

$$\begin{array}{llllll} \mathit{translate} & (dx, dy) & \mathit{im} = \mathit{im} & \cdot & \mathit{translateP} & (-dx, -dy) \\ \mathit{scale} & (sx, sy) & \mathit{im} = \mathit{im} & \cdot & \mathit{scaleP} & (1/sx, 1/sy) \\ \mathit{uscale} & s & \mathit{im} = \mathit{im} & \cdot & \mathit{uscaleP} & (1/s) \\ \mathit{rotate} & \theta & \mathit{im} = \mathit{im} & \cdot & \mathit{rotateP} & (-\theta) \end{array}$$

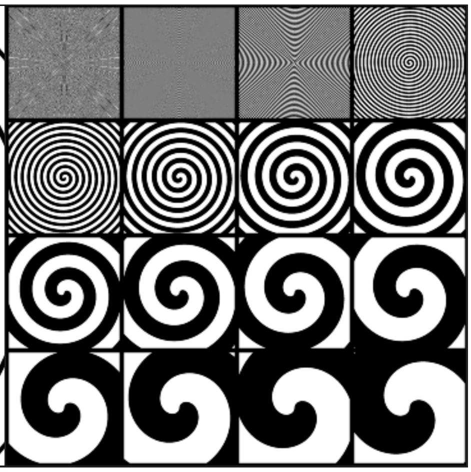
In addition to these familiar affine transforms, one can define any other kind of space-to-space function, limited only by one's imagination. For instance, here is a 'swirling' transform. It takes each point p and rotates it about

[7] Easy invertibility is one of the benefits of restricting transforms to be affine and representing them as matrices.



width = 5

Figure 7.11: *swirl* 1 *vstrip*



duration = 2, *width* = 5

Figure 7.12: $\lambda t \rightarrow swirl\ (t^2)\ xPos$

the origin by an amount that depends on the distance from p to the origin. For predictability, this transform takes a parameter r that gives the distance at which a point is rotated through a complete circle (2π radians):

$$swirlP :: Float \rightarrow Transform$$
$$swirlP\ r\ p = rotate\ (distO\ p \times 2\ \pi\ /\ r)\ p$$

$$swirl :: Float \rightarrow Filter\ \alpha \quad \text{— Image version}$$
$$swirl\ r\ im = im \cdot swirlP\ (-r)$$

Applying the *swirl* effect to *vstrip* (Figure 7.1) defined earlier results in an infinite spiral whose arms thin out away from the origin (Figure 7.11).

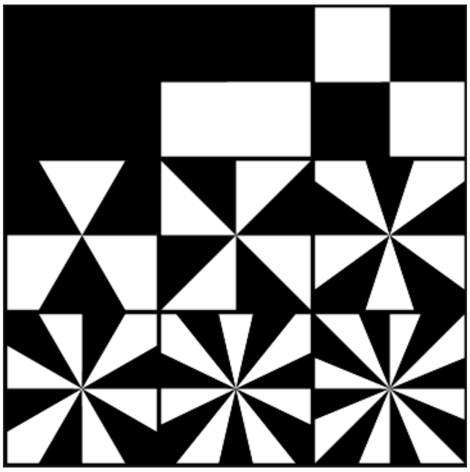
It will be useful to have compact names for transformations of colour images:

type *FilterC* = *Filter Colour*

7.6 Animation

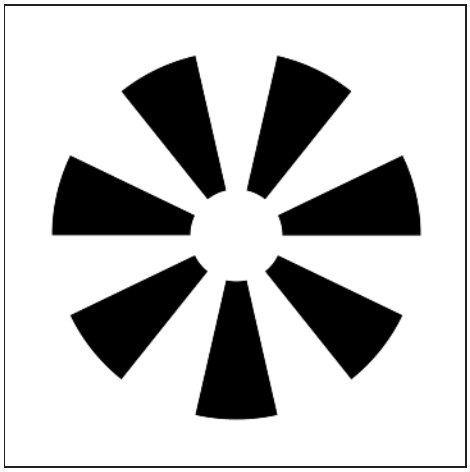
Just as an image is a function of space, an animation is a function of continuous time. This model leads to temporal resolution independence, which allows animations to be transformed in time, as easily as images are transformed in space.

type *Time* = *Float*
type *Anim* α = *Time* $\rightarrow$ *Image* α



duration = 8, *width* = 1

Figure 7.13: *radReg n* for $n = 0, \ldots, 8$



width = 2.5

Figure 7.14: *wedgeAnnulus* 0.25 7

As a simple animation example, Figure 7.12 shows what *swirl* does to the half plane *xPos* given by $x > 0$. We square time to emphasise small and large values of the *swirl* parameter.

$$\begin{aligned}
&\mathit{xPos} :: \mathit{Region}\\
&\mathit{xPos}\,(x, y) \;=\; x > 0
\end{aligned}$$

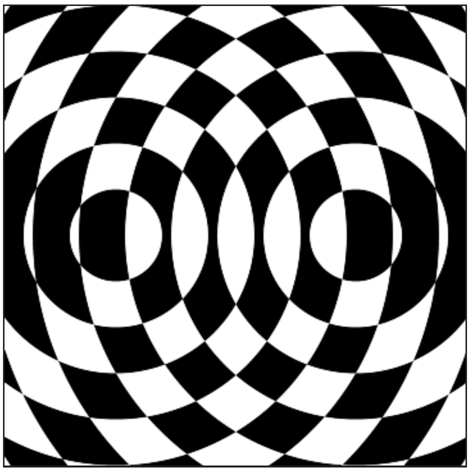
This approach to animation is adopted from Fran. See [34, 36] for many more examples.

7.7 Region algebra

Recall that 'regions' of 2D space are simply Boolean-valued images. Set operations on regions are useful and easy to define:

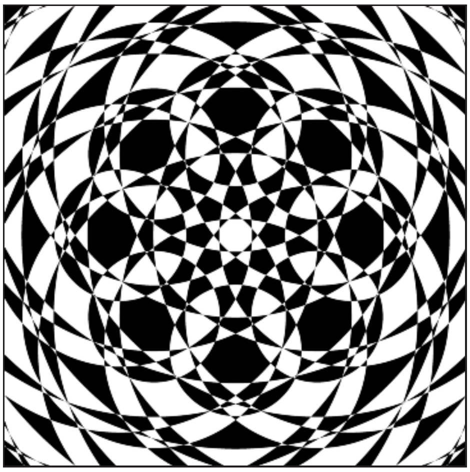
$$\begin{aligned}
&\mathit{universeR}, \mathit{emptyR} &&:: \mathit{Region}\\
&\mathit{compR} &&:: \mathit{Region} \to \mathit{Region}\\
&(\cap), (\cup), \mathit{xorR}, (\backslash) &&:: \mathit{Region} \to \mathit{Region} \to \mathit{Region}
\end{aligned}$$

$$\begin{aligned}
\mathit{universeR} &= \mathit{const\ True}\\
\mathit{emptyR} &= \mathit{const\ False}\\
\mathit{compR} &= \mathit{lift}_1\ \mathit{not}\\
(\cap) &= \mathit{lift}_2\ \mathit{and}\\
(\cup) &= \mathit{lift}_2\ \mathit{or}\\
\mathit{xorR} &= \mathit{lift}_1\ (\neq)\\
r \setminus r' &= r \cap \mathit{compR}\ r'
\end{aligned}$$



width = 10

Figure 7.15: *shiftXor* 2.6 *altRings*



width = 7

Figure 7.16: *xorgon* 8 (7/4) *altRings*

Let's see what we can do with these region operators. First, we'll build an annulus by subtracting one disk from another:

$$\begin{array}{l} \mathit{annulus} :: \mathit{Frac} \to \mathit{Region} \\ \mathit{annulus}\ \mathit{inner} \ = \ \mathit{udisk} \setminus \mathit{uscale}\ \mathit{inner}\ \mathit{udisk} \end{array}$$

Next, we'll make a region consisting of alternating infinite pie wedges (Figure 7.13, which is a simplification of Figure 7.4).

$$\begin{array}{l} \mathit{radReg} :: \mathit{Int} \to \mathit{Region} \\ \mathit{radReg}\ n \ = \ \mathit{test} \cdot \mathit{toPolar} \\ \quad \mathbf{where} \\ \qquad \mathit{test}\ (\_, \theta) \ = \ \mathit{even} \lfloor \theta \times \mathit{fromInt}\ n \,/\, \pi \rfloor \end{array}$$

Putting these two together, we get Figure 7.14.

$$\begin{array}{l} \mathit{wedgeAnnulus} :: \mathit{Float} \to \mathit{Int} \to \mathit{Region} \\ \mathit{wedgeAnnulus}\ \mathit{inner}\ n \ = \ \mathit{annulus}\ \mathit{inner} \cap \mathit{radReg}\ n \end{array}$$

The *xorR* operator is useful for creating op art[8]. For instance, Figure 7.15 is made from two copies of *altRings* (Figure 7.3), shifted in opposite directions and combined with *xorR*.

$$\begin{array}{l} \mathit{shiftXor} :: \mathit{Float} \to \mathit{Filter}\ \mathit{Bool} \\ \mathit{shiftXor}\ r\ \mathit{reg} \ = \ \mathit{reg}'\ r \ \text{`}\mathit{xorR}\text{`} \ \mathit{reg}'\ (-r) \end{array}$$

[8]A mathematically oriented form of abstract art from the 1960s, using repetition of simple forms to achieve the optical illusion of movement.

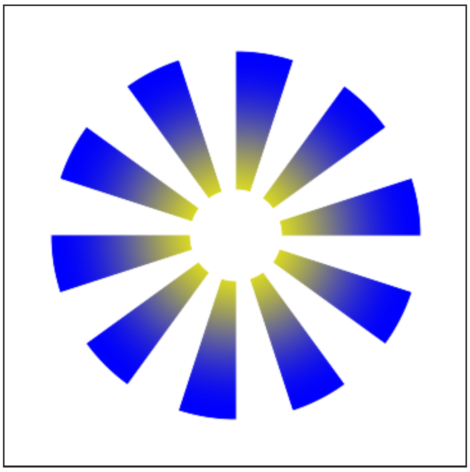


Figure 7.17: Exercise 7.9

Figure 7.18: Exercise 7.9

$$\begin{aligned}&\textbf{where}\\&\quad reg'\ d\ =\ translate\ (d, 0)\ reg\end{aligned}$$

Exercise 7.6 Generalise *shiftXor* to *xorgon*, distributing n copies of its given region around a circle of radius r and xor-ing them all together (Figure 7.16). □

Exercise 7.7 Redefine *polarChecker* (Figure 7.4) very simply by applying *xorR* to *altRings* (Figure 7.3) and *radReg* (Figure 7.13): □

Exercise 7.8 Use *xorR* and a coordinate-swapping filter to redefine *checker* (Figure 7.2) in terms of a region with alternating horizontal or vertical slabs. □

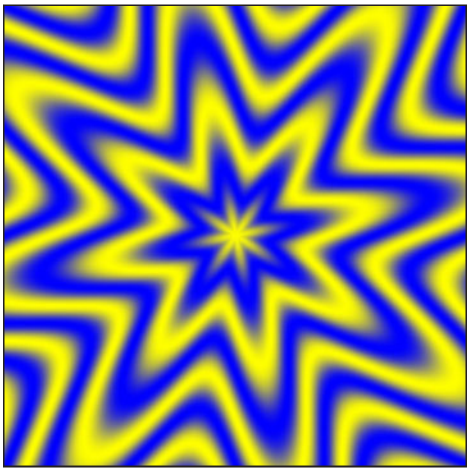
One use for regions is to crop a colour-valued image:

$$\begin{aligned}&crop :: Region \to FilterC\\&crop\ reg\ im\ =\ cond\ reg\ im\ empty\end{aligned}$$

Exercise 7.9 Express Figures 7.17 and 7.18 in terms of *wedgeAnnulus* (Figure 7.14), *ybRings* (Figure 7.7), *crop* and *swirl*. □

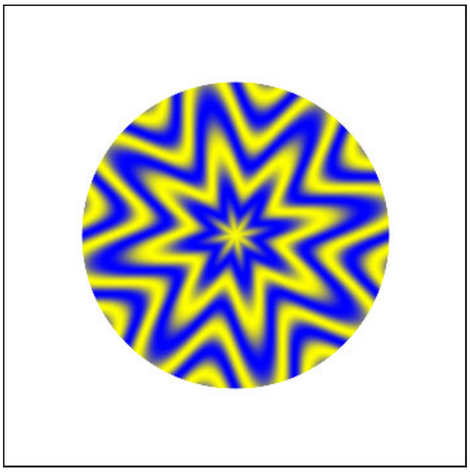
7.8 Some polar transforms

The *swirlP* function (used to define *swirl* in Section 7.5) can be expressed in terms of polar rather than rectangular coordinates



width = 10

Figure 7.19: *rippleRad* 8 0.3 *ybRings*



width = 15

Figure 7.20: Exercise 7.10

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

where the useful function *polarXf* is defined very simply:

$$polarXf\ ::\ Transform \rightarrow Transform$$
$$polarXf\ xf\ =\ fromPolar \cdot xf \cdot toPolar$$

Note that θ changes under *swirlP*, but ρ does not.

Turning things inside out

Next, let's consider a polar transform 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 7.23):

$$radInvertP\ ::\ Transform$$
$$radInvertP\ =\ polarXf\ (\lambda\ (\rho, \theta)\ \rightarrow\ (1/\rho, \theta))$$

$$radInvert\ ::\ Image\ \alpha \rightarrow Image\ \alpha$$
$$radInvert\ im\ =\ im \cdot radInvertP$$

Radial ripples

As another radial transformation, 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.

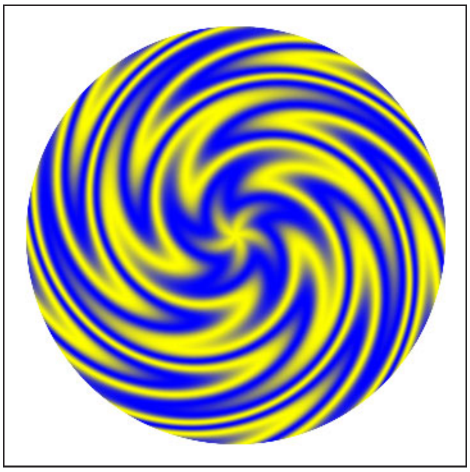


Figure 7.21: Exercise 7.10

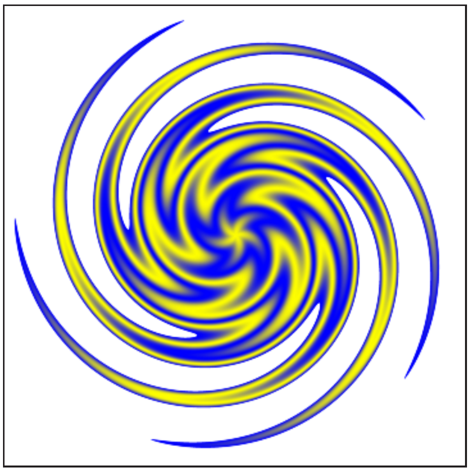


Figure 7.22: Exercise 7.10

$$
\begin{aligned}
&\mathit{rippleRadP} :: \mathit{Int} \to \mathit{Float} \to \mathit{Transform} \\
&\mathit{rippleRadP}\ n\ s \quad = \mathit{polarXf}\ \$\ \lambda\,(\rho,\theta)\ \to \\
&\qquad\qquad\qquad (\rho \times (1 + s \times \mathit{sin}\,(\mathit{fromInt}\ n \times \theta)), \theta)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{rippleRad} :: \mathit{Int} \to \mathit{Float} \to \mathit{Image}\alpha \to \mathit{Image}\alpha \\
&\mathit{rippleRad}\ n\ s\ \mathit{im} = \mathit{im}\ \cdot\ \mathit{rippleRadP}\ n\ (-s)
\end{aligned}
$$

In order to visualise the effect of *rippleRad*, apply it to *ybRings* (Figure 7.7). The result is Figure 7.19.

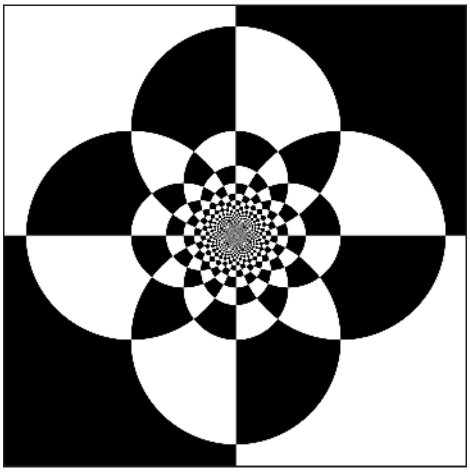
The examples so far have been infinite in size. We can also make finite ones by cropping against a region. As a convenience, we define *cropRad* as a function that crops an image to a disk-shaped region of a given radius:

$$
\begin{aligned}
&\mathit{cropRad} :: \mathit{Float} \to \mathit{FilterC} \\
&\mathit{cropRad}\ r\ = \mathit{crop}\ (\mathit{uscale}\ r\ \mathit{udisk})
\end{aligned}
$$

Exercise 7.10 Use *rippleRad*, *cropRad*, and possibly *swirl* to express Figures 7.20, 7.21 and 7.22. □

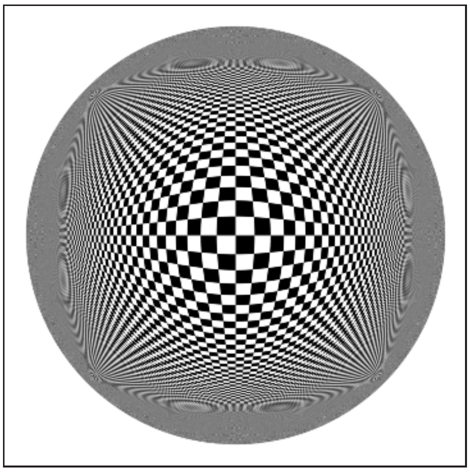
Circle limits

Figure 7.24 shows the result of squeezing the infinite *checker* image into a finite disk. Note that the spatial transformation used is essentially one-dimensional. It just moves a point closer or further from the origin, based only on its given distance.



width = 2.2

Figure 7.23: *radInvert checker*



width = 22

Figure 7.24: *circleLimit* 10 (*bwIm checker*)

$$
\begin{array}{l}
\mathit{circleLimit} :: \mathit{Float} \to \mathit{FilterC} \\
\mathit{circleLimit}\ \mathit{radius}\ \mathit{im} \ = \ \mathit{cropRad}\ \mathit{radius}\ (\mathit{im} \cdot \mathit{polarXf}\ \mathit{xf}) \\
\quad \textbf{where} \\
\qquad \mathit{xf}\ (\rho, \theta) \ = \ (\mathit{radius} \times \rho / (\mathit{radius} - \rho), \theta)
\end{array}
$$

7.9 Strange hybrids

Regions are useful for cropping images, as in *cropRad* above, but also for pointwise selection, using *cond* (Section 7.4). 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, transform the basic *xPos* region before applying selection. For convenience in constructing examples, let's define a function to select between a girl (*becky*) and her cat (*fraidy*), based on a given time-varying region:

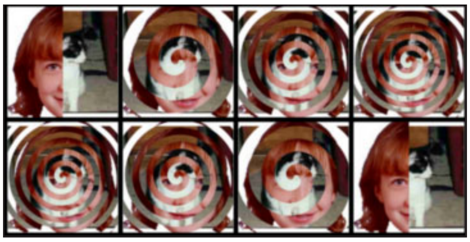
$$
\begin{array}{l}
\mathit{hybrid} :: \mathit{Anim}\ \mathit{Bool} \to \mathit{Anim}\ \mathit{Colour} \\
\mathit{hybrid}\ f\ t \ = \ \mathit{cond}\ (f\ t)\ \mathit{fraidy}\ \mathit{becky}
\end{array}
$$

Figures 7.25 through 7.27 show animations based on time-varying regions:

$$
\begin{array}{lcl}
\mathit{turningXPos}\ t & = & \mathit{rotate}\ t\ \mathit{xPos} \\
\mathit{swirlingXPos}\ t & = & \mathit{swirl}\ (10 / \mathit{sin}\ t)\ \mathit{xPos} \\
\mathit{roamingDiskt} & = & \mathit{uscale}\ 30\ (\mathit{translate}\ (\mathit{cos}\ t, \mathit{sin}\ (2 \times t))\ \mathit{udisk})
\end{array}
$$

duration = π, *width* = 120

Figure 7.25: *hybrid turningXPos*



duration = π, *width* = 120

Figure 7.26: *hybrid swirlingXPos*

Exercise 7.11 The *cond* function produces hard edges between the images being combined. Use *lerpI* (Section 7.4) to define a *softHybrid* function that shifts gradually from one image to the other. For example, Figure 7.28 shows *softHybrid* (*const* (*swirl* 10 ($wipe_2$ 75))). □

7.10 Bitmaps

In order to work with digital photos, we must reconcile two differences between our 'image' notion and the various 'bitmap' formats that can be imported.[9] Pan images have infinite domain and are continuous, while bitmaps are finite and discrete arrays, which we represent as dimensions and a subscripting function:

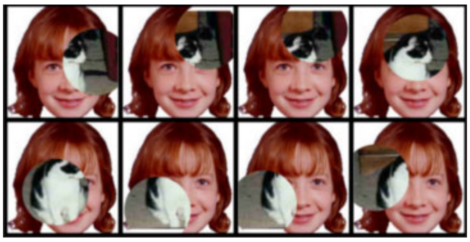
$$\mathbf{data}\ Array_2\ \alpha\ =\ Array_2\ Int\ Int\ ((Int, Int)\ \rightarrow\ \alpha)$$

That is, the value $Array_2\ n\ m\ f$ represents an array of n columns and m rows, and the valid arguments of f (and hence indices of the array) are pairs (x, y) with $0 \leqslant x < n$ and $0 \leqslant y < m$.

Rather than creating and storing an actual array of colours, each represented as a quadruple of floating point numbers, conversion from the file representation (typically 1, 8, 16, or 24 bits per pixel) 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 defined below. Sample points outside of the array's rectangular region are mapped to the invisible colour. 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 good results, Pan uses bilinear interpolation (using the function *bilerpC*, as defined in Section 7.3) to calculate

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



duration = π, *width* = 120

Figure 7.27: *hybrid roamingDisk*

width = 120

Figure 7.28: Exercise 7.11

a weighted average of the four nearest neighbours. That is, given any sample point *p*, we find the four pixels nearest to *p* and bilerp their four colours, using the position of *p* relative to the four pixels to determine the weights. (Note that *wx* and *wy* in the definition below are fractions.)

$$
\begin{array}{l}
\mathit{bilerpArray}_2 :: ((\mathit{Int}, \mathit{Int}) \to \mathit{Colour}) \to \mathit{ImageC} \\
\mathit{bilerpArray}_2\ \mathit{sub}\ (x, y) = \\
\quad \mathbf{let} \\
\qquad i = \lfloor x \rfloor;\ \mathit{wx} = x - \mathit{fromInt}\ i \\
\qquad j = \lfloor y \rfloor;\ \mathit{wy} = y - \mathit{fromInt}\ j \\
\quad \mathbf{in} \\
\quad \mathit{bilerpC}\ (\mathit{sub}\ (i, j\phantom{{}+1}))\ (\mathit{sub}\ (i+1, j\phantom{{}+1})) \\
\qquad\qquad (\mathit{sub}\ (i, j+1))\ (\mathit{sub}\ (i+1, j+1)) \\
\qquad\qquad (\mathit{wx}, \mathit{wy})
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{reconstruct} :: \mathit{Array}_2\ \mathit{Colour} \to \mathit{ImageC} \\
\mathit{reconstruct}\ (\mathit{Array}_2\ w\ h\ \mathit{sub}) = \\
\quad \mathit{move}\ (-\ \mathit{fromInt}\ w\ /\ 2, -\ \mathit{fromInt}\ h\ /\ 2) \\
\qquad\quad (\mathit{crop}\ (\mathit{inBounds}\ w\ h)\ (\mathit{bilerpArray}_2\ \mathit{sub}))
\end{array}
$$

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

$$\begin{array}{l} \mathit{inBounds} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Region} \\ \mathit{inBounds}\; w\; h\; (x, y) \;=\; 0 \leqslant x \wedge x \leqslant \mathit{fromInt}\; (w - 1) \;\wedge \\ \qquad\qquad\qquad\qquad\quad 0 \leqslant y \wedge y \leqslant \mathit{fromInt}\; (h - 1) \end{array}$$

7.11 Chapter notes

This examples in this chapter 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 mine.

I am grateful to Sigbjørn Finne and Oege de Moor for their collaboration on the implementation of Pan and for many fruitful discussions.

Jerzy Karczmarczuk independently developed a 'texture generation' system called *Clastic* [79] based on the same essential insights as in Pan. He went on to create images based on weaving, noise, solid textures, and tesselation.

Peter Henderson began the game of functional *geometry* for image synthesis [52]. Since then there have been several other such libraries [87, 140, 7, 1, 38, 36]. Many or all of these libraries are based on a spatially continuous model, but unlike Pan, none has addressed the general notion of images. A similar remark applies to the various 'vector-based' 2D APIs and file formats.

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 [57]. 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 [90]. 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' [63] used characteristic functions to represent regions, in essentially the same formulation as Pan (Section 7.7). Those regions were not used for visualisation, nor were they generalised to range types other than Boolean. Paul Hudak also used regions for graphics [61]. 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) [118]. He did not exploit Lisp's support for higher-order functional programming in composing image functions.

Bibliography

[1] Kavi Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, January 1994.

[2] Lex Augusteijn. Sorting morphisms. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, 1999.

[3] Lennart Augustsson. Cayenne: A language with dependent types. *SIGPLAN Notices*, 34(1):239–250, January 1999.

[4] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, Berlin, 1999.

[6] Denis Baggi. *Computer-Generated Music*. IEEE Computer Society Press, Las Alamitos, CA, 1992.

[7] Joel F. Bartlett. Don't fidget with widgets, draw! Technical Report 6, DEC Western Digital Laboratory, May 1991.

[8] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Conference*, pages 307–314, 1968.

[9] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[10] A. S. Bhandal, V. Considine, and G. E. Dixon. An array processor for video picture motion estimation. In J. McCanny, J. McWhirter, and E. Swartzlander, editors, *Systolic Array Processors*, pages 369–378. Prentice-Hall International, 1989.

[11] Richard Bird. An introduction to the theory of lists. In M. Broy, editor, *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Caculi of Discrete Design*. Springer Verlag, June 1987.

[12] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[13] Richard Bird and John Hughes. The alpha–beta algorithm: an exercise in program transformation. *Information Processing Letters*, 24(1):53–57, January 1987.

[14] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984. Addendum in 7(3), p. 490–492.

[15] Richard S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, 2nd edition, 1998.

[16] Richard S. Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming*. Springer-Verlag, 2002. To appear.

[17] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*. ACM, 1998.

[18] Phelim Boyle, Mark Broadie, and Paul Glasserman. Monte Carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21:1267–1321, 1997.

[19] Paul Caspi, Daniel Pilaud, Nicholas Halbwachs, and John A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, 1987.

[20] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel M.T. Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, October 2002.

[21] Olaf Chitil. Pretty printing with lazy dequeues. In *ACM SIGPLAN Haskell Workshop*, pages 183–201, Firenze, Italy, 2001. Universiteit Utrecht UU-CS-2001-23.

[22] Seonghun Cho and Sartaj Sahni. Weight biased leftist trees and modified skip lists. In *International Computing and Combinatorics Conference*, pages 361–370, June 1996.

[23] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[24] Koen Claessen and John Hughes. Testing Monadic Code with QuickCheck. In *Haskell Workshop*. ACM SIGPLAN, 2002.

[25] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computer Science Conference*, pages 62–73, Phuket, Thailand, 1999. ACM SIGPLAN.

[26] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer Verlag, second edition, 1984.

[27] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, 2001.

[28] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing: a simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

[29] Clark Allan Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Computer Science Department, Stanford University, February 1972. Available as STAN-CS-72-259.

[30] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, November 1998.

[31] Olivier Danvy, Morten Rhiger, and Kristoffer H. Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, November 2001.

[32] Oege de Moor and Ganesh Sittampalam. Generic program transformation. In *Third International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 116–149. Springer-Verlag, 1998.

[33] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269:135–162, 2001.

[34] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May/June 1999. Special Section: Domain-Specific Languages (DSL).

[35] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 2001. To appear.

[36] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, 1997.

[37] Levent Erkök and John Launchbury. Recursive monadic bindings. In *International Conference on Functional Programming*, pages 174–185, 2000.

[38] Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995.

[39] Alexandre Frey, Gérard Berry, Patrice Bertin, François Bourdoncle, and Jean Vuillemin. Jazz. Available from `http://www.exalead.com/jazz`, 1998.

[40] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* MIT Press, second edition, 2001.

[41] Jeremy Gibbons. *Algebras for Tree Algorithms.* D. Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.

[42] Jeremy Gibbons. Deriving tidy drawings of trees. *Journal of Functional Programming*, 6(3):535–562, 1996.

[43] Jeremy Gibbons. A pointless derivation of radixsort. *Journal of Functional Programming*, 9(3):339–346, 1999.

[44] Jeremy Gibbons. Calculating functional programs. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 148–203. Springer-Verlag, 2002.

[45] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *International Conference on Functional Programming*, pages 273–279, September 1998.

[46] Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232, 1993.

[47] Carlos Gonzalía. Análisis asintótico amortizado en lenguajes funcionales perezosos. In *Latin-American Conference on Functional Programming*, October 1997.

[48] Nicholas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 83–96. Springer Verlag, 1993.

[49] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[50] Richard Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[51] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. Curry: A truly functional logic language. In *ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[52] Peter Henderson. Functional geometry. In *ACM Symposium on LISP and Functional Programming*, pages 179–187, 1982.

[53] Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Symposium on Principles of Programming Languages*, pages 119–132, January 2000.

[54] Ralf Hinze. Functional Pearl: Formatting: a class act. *Journal of Functional Programming*, 2002. To appear.

[55] Thomas Ho and Sang-Bin Lee. Term Structure Movements and Pricing Interest Rate Contingent Claims. *Journal of Finance*, 41:1011–1028, 1986.

[56] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, New York, 1979.

[57] Gerard J. Holzmann. *Beyond Photography — the Digital Darkroom*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[58] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, December 1996.

[59] Paul Hudak. Haskore music tutorial. In *Second International School on Advanced Functional Programming*, pages 38–68. Springer Verlag, LNCS 1129, August 1996.

[60] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[61] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[62] Paul Hudak and Jonathan Berger. A model of performance, interaction, and improvisation. In *Proceedings of International Computer Music Conference*. International Computer Music Association, 1995.

[63] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs Awk vs ...: An experiment in software prototyping productivity. Technical report, Yale, 1994.

[64] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation: An algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[65] John Hughes. A novel representation of lists and its application to the function 'reverse'. *Information Processing Letters*, 22:141–144, 1986.

[66] John Hughes. The design of a pretty-printer library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer, 1995.

[67] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[68] MIDI 1.0 detailed specification: Document version 4.1.1, February 1990.

[69] Michael A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[70] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 470–482. ACM Press, January 1997.

[71] Steven Johnson. *Synthesis of Digital Designs from Recursion Equations*. The ACM Distinguished Dissertation Series, The MIT Press, 1984.

[72] Geraint Jones and Mary Sheeran. Collecting butterflies. Technical Monograph PRG-91, Oxford University Computing Laboratory, Programming Research Group, February 1991.

[73] Geraint Jones and Mary Sheeran. The study of butterflies. In Graham Birtwistle, editor, *Proc. 4th Banff Workshop on Higher Order*. Springer Workshops in Computing, 1991.

[74] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North Holland, 1992.

[75] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 208–232. Springer, 1993.

[76] Geraint Jones and Mary Sheeran. Designing Arithmetic Circuits by Refinement in Ruby. *Science of Computer Programming*, 22(1-2), 1994.

[77] Simon Peyton Jones. Haskell pretty-printer library. `http://www.haskell.org/libraries/#prettyprinting`, 1997.

[78] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[79] Jerzy Karczmarczuk. Functional approach to texture generation. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2002.

[80] Donald E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[81] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[82] D. Lahti. Applications of a functional programming language to hardware synthesis. Master's thesis, UCLA, 1980.

[83] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. Available from `http://research.microsoft.com/~simonpj/papers/hmap/`, 2002.

[84] John Launchbury, Jeff Lewis, and Byron Cook. On embedding a microarchitecture design language within Haskell. In *International Conference on Functional Programming*. ACM, 1999.

[85] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Functional Programming Languages and Computer Architecture*, pages 314–323. ACM Press, 1995.

[86] Daan Leijen and Erik Meijer. Domain-specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, October 1999. USENIX Association.

[87] Peter Lucas and Stephen N. Zilles. Graphics in an applicative context. Technical report, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, July 8 1987.

[88] Wayne Luk. Systematic serialisation of array-based architectures. *Integration, the VLSI Journal*, 14(3), February 1993.

[89] Wayne Luk, Geraint Jones, and Mary Sheeran. Computer-based tools for regular array design. In J McCanny, J McWhirter, and E Swartzlander, editors, *Systolic Array Processors*, pages 589–598. Prentice-Hall International, 1989.

[90] John Maeda. *Design By Numbers*. MIT Press, May 1999.

[91] John Matthews and John Launchbury. Elementary microarchitecture algebra. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 333–360. Springer, 1999.

[92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[93] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[94] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Functional Programming Languages and Computer Architecture*, 1995.

[95] Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.

[96] Marek Musiela and Marek Rutkowski. *Martingale Methods in Financial Modelling*. Springer, 1997.

[97] John O'Donnell. Hydra: Hardware description in a functional language using recursion equations and higher order combining forms. In G. J.

Milne, editor, *The Fusion of Hardware Design and Verification*, pages 363–382. North-Holland, 1988.

[98] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 195–214. Springer-Verlag, 1996.

[99] Chris Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, October 1995.

[100] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, 1995.

[101] Chris Okasaki. The role of lazy evaluation in amortized data structures. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–72, May 1996.

[102] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[103] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In Richard S. Bird and Lambert Meertens, editors, *Algorithmic Languages and Calculi*, pages 76–106. Chapman and Hall, 1997.

[104] Derek Oppen. Pretty-printing. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[105] Dorab Patel, Martine D. F. Schlag, and Milos D. Ercegovac. νFP: An environment for the multi-level specification, analysis, and synthesis of hardware algorithms. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 238–255. Springer-Verlag, 1985.

[106] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[107] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, October 1997.

[108] Vaughan Pratt. *Shellsort and Sorting Networks.* PhD thesis, Stanford University, 1971. Also Garland, New York, 1979.

[109] Daniel Revuz and Marc Yor. *Continuous Martingales and Brownian Motion.* Springer, 1991.

[110] Curtis Roads, editor. *The Music Machine (Selected Readings from Computer Music Journal).* MIT Press, Cambridge, MA, 1989.

[111] Edward Rothstein. *Emblems of Mind: The Inner Life of Music and Mathematics.* Times Books, New York, 1995.

[112] Robert Sedgewick. Analysis of Shellsort and related algorithms. In *European Symposium on Programming*, 1996.

[113] Eleanor Selfridge-Field, editor. *Beyond MIDI (The Handbook of Musical Codes)*. MIT Press, Cambridge, MA, 1997.

[114] Silvija Seres. *The Algebra of Logic Programming*. D.Phil., Programming Research Group, University of Oxford, 2001.

[115] Mary Sheeran. μFP, an algebraic VLSI design language. D.Phil., Programming Research Group, Oxford University, 1983.

[116] Mary Sheeran. Puzzling permutations. In P. Trinder, editor, *Glasgow Functional Programming Workshop*, 1996.

[117] Donald L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[118] Karl Sims. Artificial evolution for computer graphics. *ACM Computer Graphics*, 25(4):319–328, July 1991.

[119] Ganesh Sittampalam and Oege de Moor. Higher-order pattern matching for automatically applying fusion transformations. In O. Danvy and A. Filinski, editors, *Second Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 198–217. Springer-Verlag, 2001.

[120] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, February 1986.

[121] Alvy Ray Smith. Image compositing fundamentals. Technical Report Technical Memo #4, Microsoft, July 1995. `http://www.alvyray.com/Memos`.

[122] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: An efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, 1995.

[123] J. Michael Spivey. Unification: A case-study in data refinement,. *Formal Aspects of Computing*, 7(2):158–168, 1995.

[124] J. Michael Spivey. Combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, 2000.

[125] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

[126] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996.

[127] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[128] Hervé Touati and Mark Shand. PamDC: a C++ library for the simulation and generation of Xilinx FPGA designs. Available from `http://research.compaq.com/SRC/pametta/PamDC.pdf`, 1999.

[129] David A. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, 1986.

[130] Arie van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[131] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, 2000.

[132] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types.* PhD thesis, University of Tartu, 2000.

[133] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. 9th Nordic Workshop on Programming Theory.

[134] Jean Vuillemin. On circuits and numbers. *IEEE Transactions on Computers*, 43:8:868–879, 1994.

[135] Jean Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé Touati, and Philippe Boucard. Programmable Active Memories: the Coming of Age. *IEEE Trans. on VLSI*, 4(1), March 1996.

[136] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[137] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[138] Philip L. Wadler. How to replace failure by a list of successes. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, 1985.

[139] Paul Willmot, Jeff N. Dewynne, and Sam D. Howison. *Option Pricing: Mathematical Models and Computation.* Oxford Financial Press, 1993.

[140] Stephen N. Zilles, Peter Lucas, T.M. Linden, Jeff B. Lotspiech, and A.R. Harbury. The Escher document imaging model. In *ACM Conference on Document Processing Systems*, pages 159–168, December 5–9 1988.