Functional Geometry

Peter Henderson
Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
p.henderson@ecs.soton.ac.uk
http://www.ecs.soton.ac.uk/~ph

October, 2002

Abstract. An algebra of pictures is described that is sufficiently powerful to denote the structure of a well-known Escher woodcut, Square Limit. A decomposition of the picture that is reasonably faithful to Escher's original design is given. This illustrates how a suitably chosen algebraic specification can be both a clear description and a practical implementation method. It also allows us to address some of the criteria that make a good algebraic description.

Keywords: Functional programming, graphics, geometry, algebraic style, architecture, specification.

1. Background

In 1982, when the original version of this paper (6) was written, computers, computer science and functional programming were all a lot less powerful than they are today. The idea that one could write an algebraic description, embed it in a functional program, and execute it directly was not new. But it was not then considered a practical programming technique. Now we know better and many examples exist of how practical it can be to simply write denotations of what is to be constructed, rather than to write algorithmic descriptions of how to perform the construction.

A picture is an example of a complex object that can be described in terms of its parts. Yet a picture needs to be rendered on a printer or a screen by a device that expects to be given a sequence of commands. Programming that sequence of commands directly is much harder than having an application generate the commands automatically from the simpler, denotational description. Others have shown that it can be practical to directly execute denotations in other complex domains such as music (7), animation (3, 8, 11) and documents (9). Others have also developed the application of these ideas to more complex pictures (1, 2, 4). It is appropriate now to draw some general conclusions from those studies. There is no need to have read the 1982 Functional Geometry

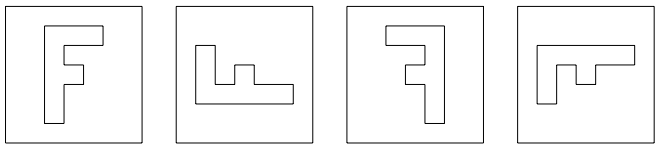

Figure 1. The pictures `f`, `rot(f)`, `flip(f)` and `rot(flip(f))`

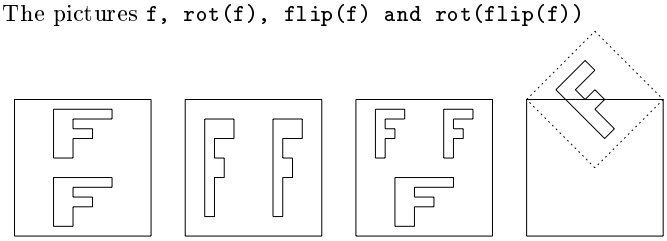

Figure 2. `above(f,f)`, `beside(f,f)`, `above(beside(f,f),f)` and `rot45(f)`

paper (6) in order to understand this one, but for comparison that earlier paper is available at the URL given in the references.

2. Basic Operations

The operations we will use to build pictures will be defined precisely in Section 5. Here we will define them informally. The algebra we will use will allow us to place these pictures next to each other in frames. The frames on which the pictures are based will be used to position each component. The operations we will define are a mixture of general operations and one which is more specialised, in a way that will become clear when we study the Escher woodcut in the next section.

Consider the four pictures shown in Figure 1. The leftmost picture is the outline of a letter. It is located within a frame, but we do not consider the frame to be part of the picture. Rather it shows the extent of the picture. If we say that `f` denotes the leftmost picture, then we denote by `rot(f)` the same picture rotated anticlockwise through a right angle. Consequently, `rot(rot(f))` would denote that the picture rotated through a full 180 degrees. When pictures are rotated, we will always rotate them in a standard, anticlockwise direction.

We denote by `flip(f)` the picture `f` flipped through its vertical centre axis. Figure 1 demonstrates the use of `rot` and `flip`. The abstraction which we have used, the object that we are referring to as a picture, has content made up of some smaller graphical objects (here, just line segments). The position, size and orientation of the picture is not part of its content. This is an essential abstraction that makes the

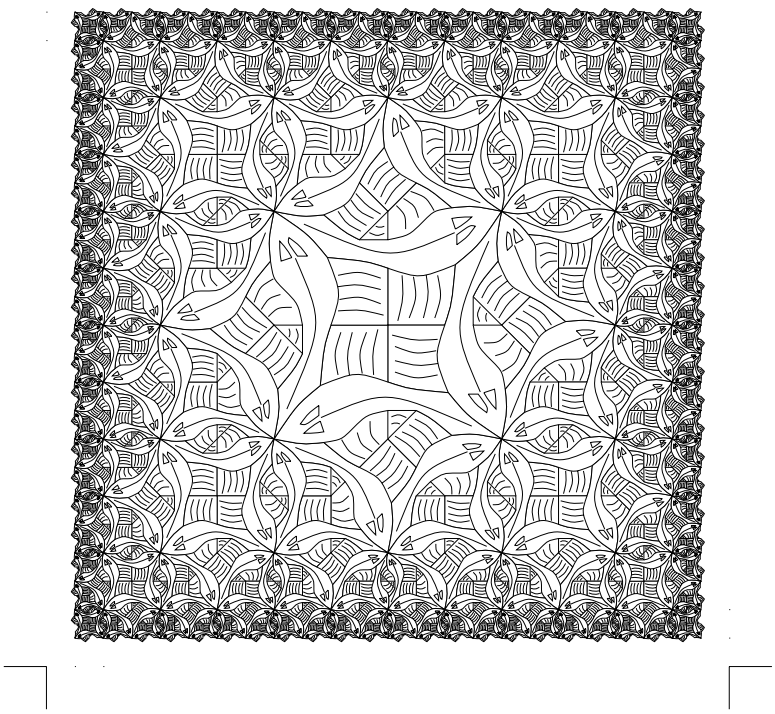

Figure 3. Square Limit to depth 3

description of pictures of the sort shown here very much simpler than they might otherwise be. The frame in which a picture is placed will be referred to as its locating box.

So far we have only used the ability to rotate and flip pictures within the box upon which they are based. Much more interesting is the ability to lay these pictures next to each other in order to form more complex pictures. We introduce two operations, `above` and `beside`, that respectively place pictures adjacent to each other vertically and horizontally. These constructions are shown in Figure 2. The construction `above(p,q)` is the picture that has `p` in the upper half of its locating box and `q` in the lower half. Similarly, the construction `beside(p,q)` is the picture that has `p` in the left half of its locating box and `q` in the right half.

The rightmost picture in Figure 2 shows the most elaborate of the operations we will need. The picture `rot45(f)` is shown superimposed with the outline of its original locating box. Unlike `rot` which rotates a picture about its centre, through a right angle, `rot45` rotates a picture about its top left corner, through 45 degrees, also anticlockwise. It also

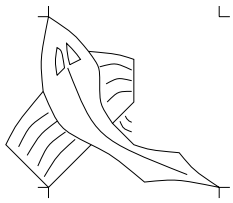

Figure 4. The basic fish

reduces the picture is size by $\sqrt{2}$, so that the diameter of its original locating box lies along the top edge of the rotated picture's locating box. This operation is defined precisely in Section 5. It is rather specific to the Escher picture we are going to construct.

One final operation `over(p,q)` overlays one picture `p` directly on top of the other picture `q`, so that the boxes on which they are based are collocated. We will see examples of that in the next section.

3. The Escher Woodcut - Square Limit

Escher's picture of interlocking fishes is reproduced in outline in Figure 3. The original is of course more elaborate in terms of its shading and various other artistic properties. The structure is all that interests us here. Each fish in the picture has the same basic shape, reoriented and resized in such a way that it neatly interlocks with its neighbours. Escher made many such pictures. This picture is of reasonable complexity compared with his others and presents a significant challenge. It was chosen because the book in which it is reproduced (10) also contains some of Escher's early drafts and sketches for it, and in particular a geometric presentation of the tessellation he was planning to use.

In the original 1982 paper I chose to start building the picture from smaller parts, rather than from a whole fish. The reasons for this will be discussed in Section 4. It seems clear that Escher did not do this, again for reasons I will discuss in that Section. Rather, he designed a single fish with the necessary interlocking features that establish the tiling property that the picture requires. There are lessons here for how we think about structuring computations and how we design software. These lessons will be returned to in Section 6.

The basic fish is shown in Figure 4. The fish is drawn inside a locating box whose corners have been indicated with chevrons (not part of the picture).

Suppose that `fish` denotes the basic fish. Figure 5 is constructed using `over(fish, rot(rot(fish)))`. This construction exemplifies one constraint on the design of the fish, that two fish laid side to side like

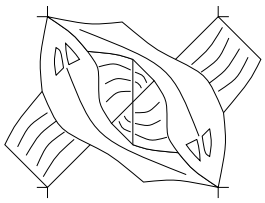

Figure 5. The basic fish, twice

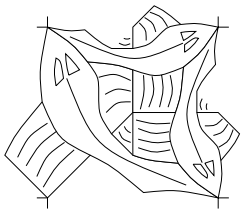

Figure 6. The basic fish, three times. The tile `t`.

this must fit each other. As an artist, Escher must have spent some time experimenting with different curves in order to achieve the appearance that he wanted. What I have done is to read off his coordinates from the hand drawn prototypes published in (10).

More remarkably, three fish fit together in a triangle, as in Figure 6. The construction for this picture is

```
t = over(fish, over(fish2, fish3))
```

where

```
fish2 = flip(rot45(fish))
fish3 = rot(rot(rot(fish2)))
```

Here we see that `fish2`, which is at the top of the picture, is obtained by rotating the basic fish anticlockwise through 45 degrees using `rot45` and then flipping it through its vertical centre axis using `flip`. Recall that `rot45` scales the picture as described in the previous Section. The third fish `fish3` is obtained from `fish2` by rotating it clockwise through a right angle, obtained here by making three anticlockwise rotations.

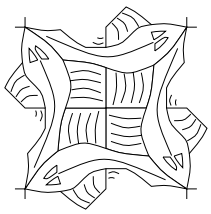

Figure 7. The basic fish, four times. The tile `u`.

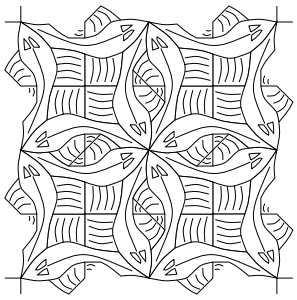

Figure 8. `quartet(u,u,u,u)`

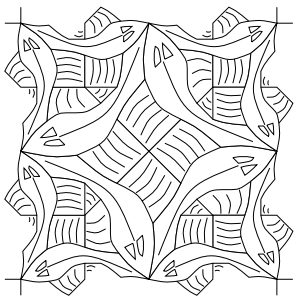

Figure 9. `v = cycle(rot(t))`

In fact the construction in Figure 6 gives us the first of two square tiles that we will use to construct the final picture. We call this tile `t`. The second square tile we will need is `u`. This tile is shown in Figure 7. Its construction is

```
u = over(over(fish2, rot(fish2)),
         over(rot(rot(fish2)), rot(rot(rot(fish2)))))
```

Basically, we take the 45 degree rotated fish from earlier (that has conveniently reduced in size by $\sqrt{2}$) and overlay four copies of it. This new interlocking is a third constraint on the design of the basic fish by Escher. In fact, the constraints that we have depicted in Figures 5, 6 and 7 induce lower level constraints upon the line segments that constitute the outline of the fish. In section 4 we will show how the outline of the fish is the same three-section line segment reproduced four times.

We begin to see how these tiles fit together in Figure 8, which is described by `above(beside(u,u),beside(u,u))`. In fact the construction in Figure 8 uses a utility function that we define as follows

```
quartet(p,q,r,s) = above(beside(p,q),beside(r,s))
```

Another utility function that we will make use of is

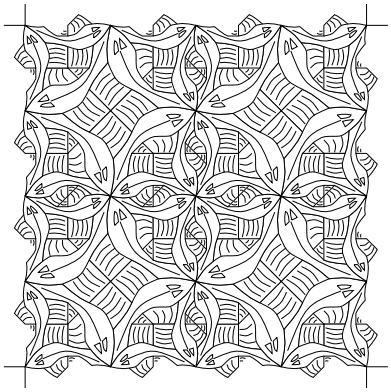

Figure 10. `quartet(v,v,v,v)`

```
cycle(p) = quartet(p, rot(p),
                   rot(rot(p)), rot(rot(rot(p))))
```

We can see this function deployed in Figure 9. This is probably the first construction that begins to look a bit like Escher's eventual woodcut, having an arrangement of fish that includes resizing and reorientation. It doesn't, however, quite demonstrate all the constraints in use. That is remedied in Figure 10, which should be self explanatory. Now we are in a position to begin building the final picture. The remainder of the construction defines a pair of equations that capture the essence of the woodcut.

The constructions we are about to make were obtained by detailed inspection of Escher's original picture. They are not the only possible decomposition of that picture but, to a computer scientist, they are the most obvious.

First we construct a component that is going to appear on the side of the final picture. Figs. 11 and 12 show the first two levels of this construction `side1` and `side2`. These constructions are defined by

```
side1 = quartet(blank,blank,rot(t),t)
side2 = quartet(side1,side1,rot(t),t)
```

and as such are specific instances of the general construction

```
side[n] = quartet(side[n-1],side[n-1],rot(t),t)
```

where `side[0] = blank`.

If we are using a lazy language, we can simply write

```
side = quartet(side,side,rot(t),t)
```

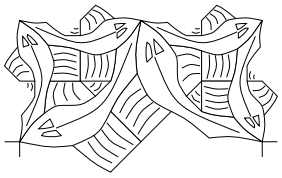

Figure 11. `side1 = quartet(blank,blank,rot(t),t)`

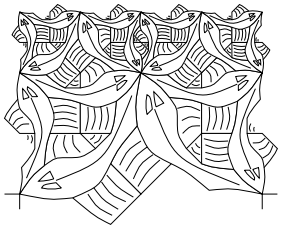

Figure 12. `side2 = quartet(side1,side1,rot(t),t)`

and rely upon the code that eventually draws the picture only reaching a finite distance into this infinite object. It only needs to build the parts of this object that are large enough to be visible on the output device.

The second construction we need is shown in Figs. 13 and 14. Here we have the parts which will comprise the corners of the final picture, to depths 1 and 2 respectively. They are defined by

```
corner1 = quartet(blank,blank,blank,u)
corner2 = quartet(corner1,side1,rot(side1),u)
```

Careful inspection of these two figures will confirm that this is indeed their description. Indeed, it was these precise descriptions which generated the pictures as they appear here.

These two equations are specific instances of the equation

```
corner[n] = quartet(corner[n-1],side,rot(side),u)
```

where we have assumed both `corner[0]` and `side[0]` are `blank`. Again, a lazy language would allow us to write

```
corner = quartet(corner,side,rot(side),u)
```

In order to build the final picture as shown in Figure 3, we need to be able to construct an arrangement of nine tiles. We call this arrangement `nonet`. It is defined as follows.

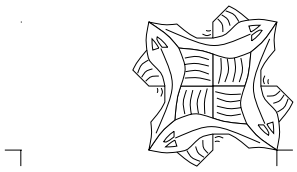

Figure 13. `corner1 = quartet(blank,blank,blank,u)`

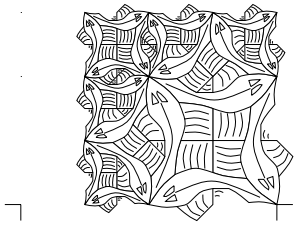

Figure 14. `corner2 = quartet(corner1,side1,rot(side1),u)`

```
nonet(p, q, r,
      s, t, u,
      v, w, x) =
            above(1,2,beside(1,2,p,beside(1,1,q,r)),
            above(1,1,beside(1,2,s,beside(1,1,t,u)),
                      beside(1,2,v,beside(1,1,w,x))))
```

Here we have used slightly more elaborate versions of `above` and `beside`. The additional parameters are numbers, m and n say, and the subpictures are arranged in the ratio $m : n$. The function `nonet` lays out its nine arguments in three rows of three.

Using these constructions we can build a simple version of Square Limit, shown in Figure 15. This takes the detail down to level 2 and has the definition

```
squarelimit2 =
    nonet(corner2, side2, rot(rot(rot(corner2))),
          rot(side2),u,rot(rot(rot(side2))),
          rot(corner2),rot(rot(side2)),rot(rot(corner2)))
```

Again, careful inspection of the picture will confirm this definition.

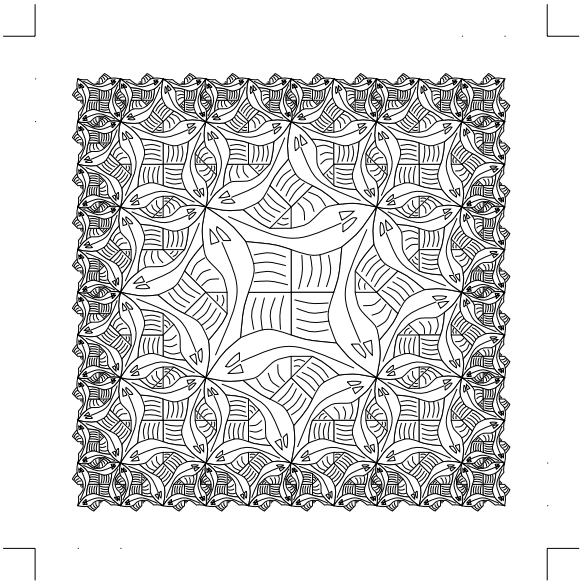

Figure 15. Square Limit to depth 2, `squarelimit2`

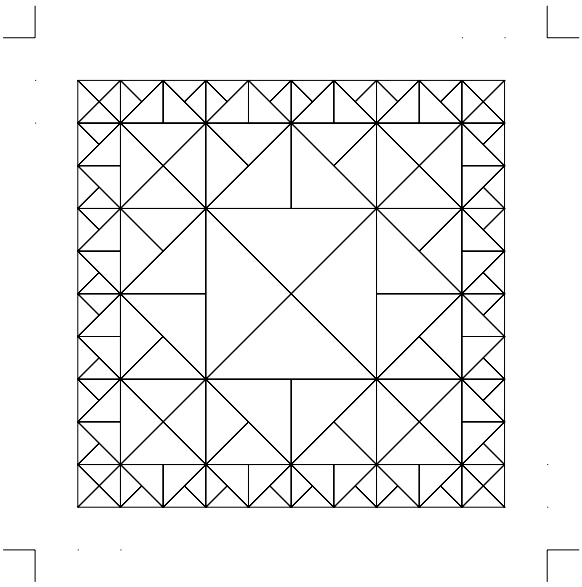

Figure 16. Drafting pattern to depth 2

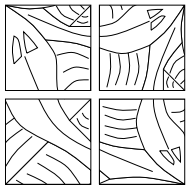

Figure 17. The original tiles, p, q, r, s, making an approximation of tile t

Figure 16 shows the pattern of triangles in which the fish for `squarelimit2` are arranged. In fact, this picture was drawn with the same equations as `squarelimit2`, but with the fish simply replaced by a triangle.

The final picture is similar, it just uses one more level of detail. It is shown in Figure 3. The general description of Square Limit is therefore

```
squarelimit =
    nonet(corner, side, rot(rot(rot(corner))),
          rot(side),u,rot(rot(rot(side))),
          rot(corner),rot(rot(side)),rot(rot(corner)))
```

4. Other Descriptions

In the 1982 version of this paper, rather than start with a whole fish, four basic square tiles similar to those shown in Figure 17 were used. The tiles `p, q, r, s` are shown in an arragement which approximately makes up the tile `t` (see Figure 6). The only difference is that the fish are not whole. If we also define `u=cycle(rot(q))`, where `q` is the top, right tile in Figure 17, we can now construct an approximation to the final picture by using this `t` and this `u`, in the equations which define `squarelimit`.

The new, square starting tiles `p, q, r, s` work because the bits of fish missing from the edges of each tile are included on the edges of the other tiles which will abut them. The edge of the finished picture will be incomplete, of course, but this may be more faithful to Escher's original as it appears in (10). However, there were other reasons for choosing these tiles, rather than a whole fish, as a starting point. Principally, they avoid the need for a rotation through any angle other than a right angle. This greatly simplifies the computation and means that the 1982 picture could be calculated using 16-bit fixed point numbers. It also avoids the problem of any line in the output being drawn twice, which on the graph plotters and high quality (dot-matrix) printers which were available at the time, had an unacceptable visual effect.

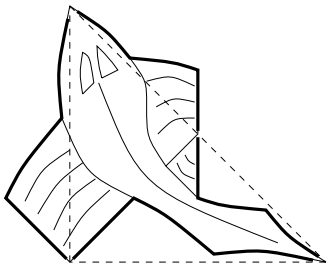

Figure 18. The four identical segments which comprise the outline of a basic fish

From the drafts Escher made of the picture, especially those in which he was experimenting with the shape of the fish, his design was of a fish which had the necessary properties for tiling the plane. The four original tiles shown in Figure 17 had no part in his thinking.

In fact, the design of the fish can be analysed more closely than we have so far attempted. Figure 18 shows the basic fish again, superimposed with four copies of a line segment which shows how its outline is made up. Also superimposed upon the fish is a triangle to show the path which these four line segments follow. Along the bottom of this picture is the basic segment. That segment is repeated, rotated through a right angle, up the left hand side of the picture, thus making up the entire left-hand side of the fish. The segment is repeated again, twice, to make up the right-hand side of the fish. These two repetitions are mirror images of the initial segment, which have been rotated 45 degrees and reduced by $\sqrt{2}$. In fact, if `s` is the segment along the bottom of the picture in Figure 18, then the entire outline is generated by

```
beside(rot(s),
       over(s, over(rot45(flip(rot(s))),
                    rot(rot(rot45(flip(rot(s))))))))
```

This little bit of geometry explains why the fishes fit together as they do.

It also explains why there is a mysterious triangle appearing in the pictures throughout this paper, from Figure 5 onwards. The triangle can be seen distinctly in Figure 18, under the right wing of the fish. To disguise it somewhat, two short lines have been placed under the wing. These can be seen most clearly in Figures 5 and 6, and have probably worried the observant reader. When the pictures are composed, these short lines appear to make up extensions to the wings of other fish, or as something (water?) in between them. In constructing the original woodcut, Escher used a little artistic licence and incorporated these triangles as continuous extensions of the fish wings. This means that

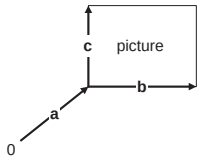

Figure 19. The basis vectors

not all the fish in the woodcut have identical outlines. In fact there are four fish, depending on whether either of the wings have been extended, or not.

This is a third problem which could be avoided by starting with the tiles in Figure 17. The triangles, which are shown in Figure 17, were actually eliminated from these tiles in the 1982 paper and hence from the final picture. Notwithstanding this visual improvement, the construction given in this paper is to be preferred, not least because the original intention of the artist was probably to tile the plane with a single fish.

5. Implementation Issues

The key conceptual idea behind Functional Geometry is that we have abstracted completely from size and absolute location. Each picture is described in terms of subpictures and their relative locations. The implementation, on rendering the picture, must work out from this description, the exact size and the exact location of each part of the picture.

We can best define this by describing what a picture denotes. Let us define a picture as a function which takes three arguments, each being two-space vectors and returns a set of graphical objects to be rendered on the output device. The vector arguments are as shown in Figure 19. The picture `p(a,b,c)` will be drawn in a box bounded by `b` and `c`, with its bottom left-hand corner at position `a`. This choice of basis vectors is not the only one we could have made, but it does make the operations we must define very easy. For example

$$over(p,q)(a,b,c) \;=\; p(a,b,c) \cup q(a,b,c)$$

defines the fact that $over(p,q)$ is the picture which includes all the objects of p with all the objects of q.

All the basic operations can be defined in this way.

$$blank(a,b,c) \;=\; \{\}$$

$$
\begin{aligned}
beside(p,q)(a,b,c) &= p(a,b/2,c) \cup q(a+b/2,b/2,c) \\
above(p,q)(a,b,c) &= p(a,b,c/2) \cup q(a+c/2,b,c/2) \\
rot(p)(a,b,c) &= p(a+b,c,-b) \\
flip(p)(a,b,c) &= p(a+b,-b,c) \\
rot45(p)(a,b,c) &= p(a+(b+c)/2,(b+c)/2,(c-b)/2)
\end{aligned}
$$

which gives a precise semantics to the operations we have used in the earlier part of this paper. For completeness we define the remaining two operations, which should be obvious

$$
\begin{aligned}
&beside(m,n,p,q)(a,b,c) = \\
&\quad p(a,b*m/(m+n),c) \cup q(a+b*m/(m+n),b*n/(m+n),c) \\
&above(m,n,p,q)(a,b,c) = \\
&\quad p(a+c*n/(m+n),b,c*m/(m+n)) \cup q(a,b,c*n/(m+n))
\end{aligned}
$$

The definition in terms of sets of graphical objects isn't quite what we want for implementation of the rendering. Rather, we want to draw each basic graphical object (here, just line segments) as we construct it and rely upon the fact that it doesn't matter if we draw the same object twice because the rendering engine will take care of that.

In fact, we want to be able to draw these pictures even if the set of graphical objects that they denote is infinite. So the implementation will need to implement some rule such as $p(a,b,c) = \{\}$, if $|b+c| < \epsilon$, where ϵ is some dimension considered too small to draw.

Of course, we haven't defined how basic pictures (such as the fish in Figure 4) should be created. In the implementation used to create the pictures in this paper, the fish is made up of about 30 bezier curves and is implemented as an object which can calculate the parameters of those beziers given the parameters a, b, c. By avoiding the use of fills, it is not necessary to be concerned with the order in which the graphical objects are rendered.

6. General Considerations

The basic idea exemplified in this paper is simple. Denotations of objects are easier to understand than algorithmic descriptions of how to build those objects. Here we have used a graphical object, the Square Limit picture, and shown that a certain decomposition of it is both a clear description of its structure and sufficient information to actually drive the device that draws the picture.

The consequence of thinking in terms of what objects denote is that the operations which we define form an algebraic system. The algebra

of pictures enjoys rules which are useful in transforming descriptions. For example

$$\begin{aligned} rot(rot(rot(rot(p)))) &= p \\ rot(above(p,q)) &= beside(rot(p), rot(q)) \\ rot(beside(p,q)) &= above(rot(q), rot(p)) \\ flip(beside(p,q)) &= beside(flip(q), flip(p)) \end{aligned}$$

There are many more rules of this sort. It seems there is a positive correlation between the simplicity of the rules and the quality of the algebra as a description tool.

The operations defined above and about which we have produced some rules are clearly generic in that they would be generally applicable to the construction of many pictures. Not so generic is the operation $rot45$, which is almost certainly specific to the construction behind Square Limit. The $rot45$ operation is not as simple as its name suggests nor as any of the other, more generic, operations. In fact it does something quite odd, rotating the picture about its top left hand corner, rather than about its centre. The consequence of this is that it doesn't enjoy simple rules. For example, while

$$rot45(rot45(p)) \neq rot(p)$$

a more complex rule, which approximates what we might have expected, is

$$\begin{aligned} &above(blank, rot45(rot45(p))) = \\ &\quad above(quartet(blank, blank, rot(p), blank), blank) \end{aligned}$$

This indicates that two $rot45$s actually move the picture p out of the box on which it is based, into a box above that. Also, since each reduces the picture by $\sqrt{2}$, their combination actually halves the size of p.

This analysis suggests that $rot45$ is not the kind of operation we would include in a general algebraic language for geometry. Rather, we would define more fundamental operations for scaling, rotation and translation and allow the user to define $rot45$ for this specific application.

Others have developed the application of these ideas to more complex pictures (1, 2, 4). In particular, Chailloux and Cousineau (2), having repeated the Square Limit construction, also do Escher's Circle Limit, based on considerably more complex transformations than are discussed here.

Algebraic structures, along these lines, have been developed for music by Hudak et al. (7) and for animation by Elliott and Hudak (3, 8),

and by Thompson (11). Each has shown that the denotation is sufficient to the construction of the eventual artifact (music, or animation).

More recently, contemporary approaches to document structure has tended to the denotational, when described with contemporary technologies such as XML. More specifically, the denotations can be manipulated with languages such as XSLT, which are essentially declarative, as Kay (9) explains. The general considerations which algebraic approaches require do seem to scale up to realistic applications, especially where the application domain is fairly uniform.

7. Conclusions

Having deconstructed Escher's woodcut Square Limit, as illustrated in (10), we have reconstructed it using a denotational metaphor. In particular, we have viewed pictures as hierarchical compositions of graphical objects and shown how some simple operations can together give an algebra powerful enough to both describe what a picture denotes and to drive an application which can draw that picture.

This idea is not new. It was published in 1982, but even then it was based on contemporary views of what was good practice in declarative systems. Since that time, the idea has been applied to many more complex domains and, as time has progressed, does seem to be one which has come to be applicable on a more practical scale.

Whether Maurits Escher would approve of the deconstruction of his elegant picture is, sadly, something we can only wonder about.

8. Acknowledgements

I am grateful to many people who have commented on earlier drafts of this paper and made useful suggestions. These have done much to improve the presentation. These helpful people include Olivier Danvy, Matthew James Henderson, Jerzy Karczmarczuk, Julia Lawall, Henning Korsholm Rohde and Peng Fei Xue. Many thanks.

The pictures were drawn by a Java program which generated postscript commands directly. The Java was written in a functional style (using static methods) so that the definitions which were executed were exactly as they appear in the paper, with the addition of the occasional semicolon.

References

Abelson, Harold, Gerald Jay Sussman and Julie Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press (1985, second edition 1996)

Chailloux, Emmanuel and Guy Cousineau. *Programming Images in ML.* Proceedings of the ACM SIGPLAN Workshop on ML and its Applications (1992)

Elliott, Conal and Paul Hudak. *Functional Reactive Animation.* Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)

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

Henderson, Peter. *Functional Programming - Application and Implementation.* Prentice-Hall, (1980)

Henderson, Peter. *Functional Geometry.* Proceedings of 1982 ACM Symposium on Lisp and Funtional Programming, 179–187, ACM, (1982) (see http://www.ecs.soton.ac.uk/~ph/funcgeo.pdf)

Hudak, Paul, Tom Makucevich, Syam Gadde and Bo Whong. *Haskore Music Notation - An Algebra of Music.* Journal of Functional Programming, vol. 6, no. 3, 465–483, (1996)

Hudak, Paul. *Modular Domain Specific Languages and Tools.* Proceedings: Fifth International Conference on Software Reuse, IEEE Computer Society Press, 134–142, (1998)

Kay, M. *XSLT Programmer's Reference.* 2nd Ed, Wrox Press Ltd., ISBN: 1861005067, (April 2001)

Locher J.L. (ed) *The World of M.C.Escher.* Harry N. Abrahams Inc., New York, ISBN 810901012, (1971)

Thompson, Simon. *A Functional Reactive Animation of a Lift using Fran.* Journal of Functional Programming, vol. 10, no. 3, 245–268, (2000)