Composing Fractals

MARK P. JONES
Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU
20000 NW Walker Road, Beaverton, OR 97006, USA

Abstract

This paper describes a simple but flexible family of Haskell programs for drawing pictures of fractals such as Mandelbrot and Julia sets. Its main goal is to showcase the elegance of a compositional approach to program construction, and the benefits of a clean separation between different aspects of program behavior. Aimed at readers with relatively little experience of functional programming, the paper can be used as a tutorial on functional programming, as an overview of the Mandelbrot set, or as a motivating example for studies in computability.

1 Introduction

The Mandelbrot set is probably one of the best known examples of a *fractal*. From a mathematical perspective, its definition seems elementary and straightforward. But attempts to visualize it—including those of Benoit Mandelbrot who, in the late-1970s (Mandelbrot, 1975; Mandelbrot, 1988), was the first to apply computer imaging to the task—reveal an amazingly intricate and attractive structure.

This paper describes some simple but flexible programs, written in Haskell (Peyton Jones, 2003), that generate pictures of the Mandelbrot set. Thanks to their elegant, compositional construction, we will see that different aspects of behavior are cleanly separated as independent concerns. For example, the picture in Figure 1 shows one view of the Mandelbrot set produced by the program in this paper, using nothing more than standard characters on a printed page to produce a pleasing image. With a few minor changes, the same basic program can be used to explore a different portion of the Mandelbrot set, to visualize a different type of fractal, or to render the resulting image using colored pixels on a graphical display.

Another interesting observation about the programs in this paper is that there are *no recursive definitions* of any kind. Instead, the code uses higher-order functions from the standard Haskell prelude to capture common patterns of computation, particularly in the case of list processing. This property was noticed only after all

```
                            ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
                        ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
                    ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
                 ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,................,,,,,,,,,,,,,,,,,
              ,,,,,,,,,,,,,,,,,,,,,,,,,,,,..........‘‘‘‘‘........,,,,,,,,,,,,,,
            ,,,,,,,,,,,,,,,,,,,,,,,,,............‘‘‘":|~""‘‘‘......,,,,,,,,,,,,
         ,,,,,,,,,,,,,,,,,,,,,,,,.............‘‘‘‘‘"~:oOo-$~"‘‘........,,,,,,,,
       ,,,,,,,,,,,,,,,,,,,,,,..............‘‘‘‘‘‘""~:;-?+-;~""‘‘‘‘.......,,,,,,
     ,,,,,,,,,,,,,,,,,,,,..............‘‘‘‘‘‘‘"""~:?>$$$$&/|:~""‘‘‘‘.......,,,,
   ,,,,,,,,,,,,,,,,,,...............‘‘‘‘‘""""~~~~:;o^$$$$$$!;:~~""""‘‘‘.....,,,
 ,,,,,,,,,,,,,,,,...............‘‘‘‘‘‘‘""~;$<<oo!$|$>{$$$$>/X!$o:::;=~"‘.....,,
,,,,,,,,,,,,,,,...........‘‘‘‘‘‘‘‘‘‘""""~~;!{$$=$$$$$$$$$$$$$$$$|$8$=o"‘‘.....,
,,,,,,,,,,,,,,........‘‘‘‘‘‘‘""""""""""~~:o||+$$$$$$$$$$$$$$$$$$$$$$O-:""‘‘....
,,,,,,,,,,,,,.......‘‘‘‘"-o~~~~~~~~~~~:;o/$$$$$$$$$$$$$$$$$$$$$$$$$$|;;~‘‘‘....
,,,,,,,,,,,......‘‘‘‘‘""~;X!--o!^-oo;;-X$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$+~"‘‘....
,,,,,,,,,,....‘‘‘‘‘‘"""~::o/$$#$$$$$$$|!?$$$$$$$$$$$$$$$$$$$$$$$$$$$$?&~"‘‘....
,,,,,,,....‘‘‘‘‘‘"""~;;;-$$$$$$$$$$$$${{$$$$$$$$$$$$$$$$$$$$$$$$$$$$$!:"‘‘‘....
,,,,,,..‘‘‘""~"~~~~::;!=$+$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$?:~"‘‘‘....
,,,,,.‘{$*@8$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$/o:~""‘‘‘....
,,,,,,..‘‘‘""~"~~~~::;!=$+$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$?:~"‘‘‘....
,,,,,,,....‘‘‘‘‘‘"""~;;;-$$$$$$$$$$$$${{$$$$$$$$$$$$$$$$$$$$$$$$$$$$$!:"‘‘‘....
,,,,,,,,,,....‘‘‘‘‘‘"""~::o/$$#$$$$$$$|!?$$$$$$$$$$$$$$$$$$$$$$$$$$$$?&~"‘‘....
,,,,,,,,,,,......‘‘‘‘‘""~;X!--o!^-oo;;-X$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$+~"‘‘....
,,,,,,,,,,,,,.......‘‘‘‘"-o~~~~~~~~~~~:;o/$$$$$$$$$$$$$$$$$$$$$$$$$$|;;~‘‘‘....
,,,,,,,,,,,,,,........‘‘‘‘‘‘‘""""""""""~~:o||+$$$$$$$$$$$$$$$$$$$$$$O-:""‘‘....
,,,,,,,,,,,,,,,...........‘‘‘‘‘‘‘‘‘‘""""~~;!{$$=$$$$$$$$$$$$$$$$|$8$=o"‘‘.....,
 ,,,,,,,,,,,,,,,,...............‘‘‘‘‘‘‘""~;$<<oo!$|$>{$$$$>/X!$o:::;=~"‘.....,,
   ,,,,,,,,,,,,,,,,,,...............‘‘‘‘‘""""~~~~:;o^$$$$$$!;:~~""""‘‘‘.....,,,
     ,,,,,,,,,,,,,,,,,,,,..............‘‘‘‘‘‘‘"""~:?>$$$$&/|:~""‘‘‘‘.......,,,,
       ,,,,,,,,,,,,,,,,,,,,,,..............‘‘‘‘‘‘""~:;-?+-;~""‘‘‘‘.......,,,,,,
         ,,,,,,,,,,,,,,,,,,,,,,,,.............‘‘‘‘‘"~:oOo-$~"‘‘........,,,,,,,,
            ,,,,,,,,,,,,,,,,,,,,,,,,,............‘‘‘":|~""‘‘‘......,,,,,,,,,,,,
              ,,,,,,,,,,,,,,,,,,,,,,,,,,,,..........‘‘‘‘‘........,,,,,,,,,,,,,,
                 ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,................,,,,,,,,,,,,,,,,,
                    ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
                        ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
                            ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

Fig. 1. A picture of the Mandelbrot Set

of the code had been written, and was never an explicit design goal. As such, it highlights the role that higher-order functions play in supporting a natural, high-level, and compositional approach to program construction.

In the interests of brevity, some familiarity with Haskell is assumed; newcomers may find it helpful to read this paper in conjunction with one of the available introductory textbooks (Bird, 1998; Thompson, 1999; Hudak, 2000).

2 What is the Mandelbrot Set?

At the simplest level, the Mandelbrot set is just a collection of points, each of which is a pair of floating point numbers.

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

There are two steps in the procedure for determining whether a given point p is a member of the Mandelbrot set (or not). In the first step, we use the coordinates of p to construct a sequence of points, *mandelbrot* p. In the second step, we examine the points in this sequence and, if they are all “fairly close” to the origin, then we conclude that p is a member of the Mandelbrot set. But if the points get further and further from the origin, then we can be sure that p is not a member of the Mandelbrot set. (The technical details will be made more precise later in the paper.)

The following function—whose simple definition stands in striking contrast to the complexity of our fractal images—is the key to the whole process:[1]

$$\begin{array}{lcl} \mathit{next} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{Point} \\ \mathit{next}\ (u,\ v)\ (x,\ y) & = & (x * x - y * y + u,\ 2 * x * y + v) \end{array}$$

If we pick a point p and another point z, then we can apply *next* p repeatedly to z to generate the following infinite sequence:

$$[z,\ \mathit{next}\ p\ z,\ \mathit{next}\ p\ (\mathit{next}\ p\ z),\ \mathit{next}\ p\ (\mathit{next}\ p\ (\mathit{next}\ p\ z)),\ \ldots$$

Building such sequences is a perfect application for the *iterate* function in the Haskell standard prelude, which relies on lazy evaluation to allow the construction of infinite lists. For the Mandelbrot set, we pick z to be the origin, $(0,\ 0)$, and we construct the sequence corresponding to a point p using the following definition:

$$\begin{array}{lcl} \mathit{mandelbrot} & :: & \mathit{Point} \to [\mathit{Point}] \\ \mathit{mandelbrot}\ p & = & \mathit{iterate}\ (\mathit{next}\ p)\ (0,\ 0) \end{array}$$

For example, if we pick p as the origin, then all of the points are the same:

$$\begin{array}{l} \mathit{mandelbrot}\ (0,\ 0) \\ \quad \Longrightarrow [(0,\ 0),\ (0,\ 0),\ (0,\ 0),\ (0,\ 0),\ (0,\ 0),\ \ldots \end{array}$$

If we start with larger coordinate values, then the numbers can grow rapidly:

$$\begin{array}{l} \mathit{mandelbrot}\ (0.5,\ 0) \\ \quad \Longrightarrow [(0.5,\ 0),\ (0.75,\ 0),\ (1.0625,\ 0),\ (1.62891,\ 0),\ (3.15334,\ 0),\ \ldots \end{array}$$

There are also cases where the trend is not so clear. For example, at first glance, the first coordinates in the following sequence seem to be increasing slowly, but steadily, at each step:

$$\begin{array}{l} \mathit{mandelbrot}\ (0.1,\ 0) \\ \quad \Longrightarrow [(0.1,\ 0),\ (0.11,\ 0),\ (0.1121,\ 0),\ (0.112566,\ 0),\ (0.112671,\ 0),\ \ldots \end{array}$$

However, if we look further down the sequence, for example, at the 100th point, which is $(0.112702,\ 0.0)$, then we see that it is still quite close to the initial points. And, if we look even further, at the 200th point, then we see that the value is unchanged at $(0.112702,\ 0.0)$. Wary of the problems that can be caused by rounding and truncation, the wise reader will always approach examples involving floating point calculations with great care. However, in this case, a quick calculation confirms that $(0.112702 * 0.112702) + 0.1 = 0.112702$, at least to the accuracy shown. It now follows that all elements in *mandelbrot* $(0.1,\ 0)$ from the 100th onwards (and possibly some before) are in fact equal. (Switching from *Float* to a double precision

[1] Readers with a mathematical background may prefer to think of the Mandelbrot set as a set of complex numbers, with values $z = x + iy$ corresponding to the points $(x,\ y)$ used here. In that setting, the *next* function has an even simpler definition as $\mathit{next}\ p\ z = z^2 + p$, where $p = u + iv$.

floating point number type like *Double* will not prevent these problems, although it would delay their appearance.) From these calculations, we can deduce that (0, 0) and (0.1, 0) are members of the Mandelbrot set, while (0.5, 0) is not.

3 The Need for Approximation

Our next task is to code up the test on *mandelbrot p* sequences so that we can determine whether the corresponding points *p* are members of the Mandelbrot set. For reasons that we describe below, it is technically impossible to write a program to carry out the necessary tests with complete accuracy. Fortunately, for the purposes of visualization, we do not need complete accuracy; a reasonable approximation will do. In fact, if our main objective is to produce attractive images, then there are significant advantages in using approximations because of the way that they allow us to use a range of different characters or colors in the pictures that we produce.

First, we need to be more precise about what is meant by saying that a point (u, v) is "fairly close" to the origin. In fact, we will say that this holds if, and only if, the point is within a distance of 10 from the origin. (The choice of the constant 10 here is somewhat arbitrary; different values will have an effect on the coloring of our images, but not on their basic form.) Using Pythagoras' theorem, this is equivalent to requiring that $\sqrt{u^2 + v^2} < 10$. Squaring both sides to avoid the square root, we can capture this condition in the definition of a predicate:

```
fairlyClose          :: Point → Bool
fairlyClose (u, v)   =  (u ∗ u + v ∗ v) < 100
```

Now we can return to the task of deciding whether a given point *p* is a member of the Mandelbrot set. To do this, we need to check that all of the points in the corresponding sequence are close to the origin. The test can be expressed very succinctly in Haskell using the prelude function *all*:

```
inMandelbrotSet     :: Point → Bool
inMandelbrotSet p   =  all fairlyClose (mandelbrot p)
```

This function checks each element of the sequence *mandelbrot p* in turn. If it encounters a point that does not satisfy the *fairlyClose* predicate, then it terminates with result *False*, and we can conclude that *p* is not a member of the Mandelbrot set. However, this computation could be quite expensive: we might have to look at many different values from the sequence before finding one for which the test fails. Worse still, until we have found such a point, we cannot be sure that our program will *ever* find one. Perhaps the very next point will be the one that we are looking for? Or perhaps, as yet unknown to the program, *p* is actually a member of the Mandelbrot set, and we will *never* find a point for which the test fails. Instead of returning a definite *True* or *False*, the *inMandelbrotSet* function will either return *False*, possibly after a long delay, or it will go into an infinite loop. What we have observed here, informally, is that our simple test for determining membership in the Mandelbrot set is not, in more formal terminology, a *computable* function.

One way to sidestep this problem is to restrict attention to some fixed number of points at the beginning of each *mandelbrot p* sequence; if all of those points are close to the origin, then chances are good that *p* is a member of the Mandelbrot set (or at least close to it). We can capture this idea with a simple modification of *inMandelbrotSet*, using *take* to select just the first *n* elements in each sequence:

$$\begin{array}{lcl} approxTest & :: & Int \to Point \to Bool \\ approxTest\ n\ p & = & all\ fairlyClose\ (take\ n\ (mandelbrot\ p)) \end{array}$$

Of course, in some cases, *approxTest* will give a wrong answer. If the first *n* points are all close to the origin, then *approxTest* will return *True*, even if the very next point would have caused the test to fail. But, by increasing the value of *n*, we can make the test as accurate as we like and still be sure that the test will always produce either a *True* or *False* result after a limited amount of computation.

For the purposes of drawing a picture, a simple Boolean result gives only one bit of information for each point *p* that is being considered as a candidate for membership in the Mandelbrot set. With the rich palette of colors or characters that are available on typical output devices, it seems a shame to restrict ourselves to monochrome images! With that in mind, and to avoid prematurely committing the code to a particular output method, let us suppose that we have a non-empty, finite list, *palette*, that contains values representing each of the different 'colors' that we might like to use in our fractal images. Instead of trying to determine whether all of the points in a sequence are *fairlyClose* to the origin, we will only count how many initial points meet this criterion, up to a finite limit (the length of the *palette*). If the first point in a sequence fails the test, then we will display it using the first entry in the palette; if the test fails when it reaches the second point, then we assign the second color from the palette; and so on. The following definition of *chooseColor* shows how this process can be described using function composition to build a simple pipeline (!! is the list indexing operator):

$$\begin{array}{lcl} chooseColor & :: & [color] \to [Point] \to color \\ chooseColor\ palette & = & (palette\,!!)\,.\,length\,.\,take\ \ n\,.\,takeWhile\ fairlyClose \\ & & \textbf{where}\ n\ \ =\ \ length\ palette - 1 \end{array}$$

Notice that this definition is polymorphic: the identifier *color* appearing in the type is a *type variable*, so it can be instantiated to different types in different settings. We will benefit from this flexibility later by using palettes made from a list of characters for images like the one in Figure 1, and palettes containing RGB color values for images to be plotted using high-resolution graphics primitives.

4 From Points to Pictures

Now we turn our attention from individual points to the construction of complete images. At a high level, and following Pan (Elliott, 2003), an image is just a mapping that assigns a *color* value to each point:

$$\textbf{type}\ \mathit{Image}\ \mathit{color} \quad = \quad \mathit{Point} \to \mathit{color}$$

Note here again that *color* is a type variable, which allows us to accommodate different types of drawing mechanisms and palettes. Indeed, the Mandelbrot set can be thought of as a value of type *Image Bool*, mapping points that are in the set to *True* and points outside it to *False*. This, of course, is precisely what we tried to do with the *inMandelbrotSet* function in the last section.

To obtain a more colorful image, we can combine a *fractal* sequence generator (such as *mandelbrot*) with a suitable *palette* using the *chooseColor* function:

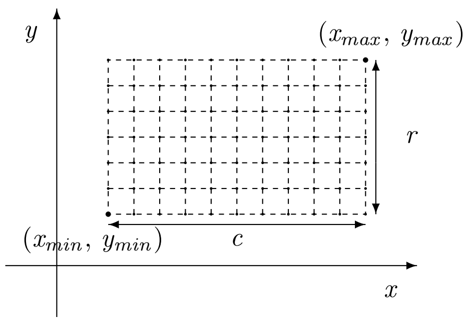
$$\begin{array}{lcl}\mathit{fracImage} & :: & (\mathit{Point} \to [\mathit{Point}]) \to [\mathit{color}] \to \mathit{Image}\ \mathit{color} \\ \mathit{fracImage}\ \mathit{fractal}\ \mathit{palette} & = & \mathit{chooseColor}\ \mathit{palette} \,.\, \mathit{fractal}\end{array}$$

This gives a function that specifies a color for every point value. On devices like a monitor or printer, however, images are produced by specifying colors only for the points on a bounded, evenly-spaced, rectangular grid of rows and columns. We will represent grids like these as lists of lists:

$$\textbf{type}\ \mathit{Grid}\ a \quad = \quad [[a]]$$

For example, a picture with r rows and c columns can be described by a list of length r, with one entry for each row, each of which is a list of length c. The values in each position will depend on the kind of picture that we are trying to produce: for example, they might be characters or pixel colors. In fact, it is also useful to work with grids containing points and with grids containing sequences, which further motivates the decision to make *Grid* a parameterized type.

Next we consider the task of constructing these grids. To start with, each of our pictures covers a range of point values, which can be described by the coordinates (x_{min}, y_{min}) of the point at the bottom left corner, and the coordinates (x_{max}, y_{max}) of the point at the top right. Of course, there are limits on the number of rows and columns that we can display on the screen at any one time, so we cannot expect to deal with all of the points in this region. Instead, we will choose an evenly spaced grid of points that covers the range with the appropriate number of rows and columns:



Each grid is determined by four parameters: the number of columns c, the number of rows r, the point (x_{min}, y_{min}) at the bottom left corner, and the point (x_{max}, y_{max}) at the top right. These can be modified to vary the detail in the final picture. For example, the simple grid above has just 7 rows and 11 columns, while the picture in Figure 1 has 37 rows and 79 columns. For a given choice of parameters, we can describe the construction of an appropriate grid of points using a function:

$$\begin{aligned} &\mathit{grid} :: \mathit{Int} \to \mathit{Int} \to \mathit{Point} \to \mathit{Point} \to \mathit{Grid\ Point} \\ &\mathit{grid}\ c\ r\ (x_{min}, y_{min})\ (x_{max}, y_{max}) \\ &\quad = [[(x, y) \mid x \leftarrow \mathit{for}\ c\ x_{min}\ x_{max}] \mid y \leftarrow \mathit{for}\ r\ y_{min}\ y_{max}] \end{aligned}$$

In constructing this grid, we need to pick c evenly spaced values for x in the range x_{min} to x_{max}, and r evenly spaced values for y in the range y_{min} to y_{max}. These two calculations are carried out in essentially the same way, so we define an auxiliary function to take care of this:

$$\begin{aligned} &\mathit{for} &&:: \mathit{Int} \to \mathit{Float} \to \mathit{Float} \to [\mathit{Float}] \\ &\mathit{for}\ n\ \mathit{min}\ \mathit{max} &&= \mathit{take}\ n\ [\mathit{min}, \mathit{min} + \mathit{delta}\ ..] \\ & && \quad \mathbf{where}\ \mathit{delta} = (\mathit{max} - \mathit{min}) / \mathit{fromIntegral}\ (n - 1) \end{aligned}$$

The only slight subtlety here is the use of *fromIntegral* to convert the integer $(n - 1)$ so that it can be used in floating point arithmetic.

Given a grid point, we can sample the image at each position to obtain a corresponding grid of colors that is ready for display. This sampling process is easy to describe using nested calls to *map* to iterate over the list of lists in the input grid:

$$\begin{aligned} &\mathit{sample} &&:: \mathit{Grid\ Point} \to \mathit{Image\ color} \to \mathit{Grid\ color} \\ &\mathit{sample\ points\ image} &&= \mathit{map}\ (\mathit{map\ image})\ \mathit{points} \end{aligned}$$

We now have all of the pieces that we need to draw pictures of fractals. In each case, we use a *fractal* function (such as *mandelbrot*) and a suitable *palette* to build an image; we sample the image on a given grid of *points*; and we *render* the resulting grid of colors. We can capture this pattern very easily with a higher-order function:

$$\begin{aligned} &\mathit{draw\ points\ fractal\ palette\ render} \\ &\quad = \mathit{render}\ (\mathit{sample\ points}\ (\mathit{fracImage\ fractal\ palette})) \end{aligned}$$

To a large degree, each of the parameters here can be varied independently of the others. Of course, the type of colors that we include in our *palette* must be compatible with the function that will be used to *render* the final image. This is captured naturally by a shared type variable, *color*, in the type of *draw*:

$$
\begin{aligned}
draw \; :: \; & Grid\ Point \\
& \rightarrow (Point \rightarrow [Point]) \\
& \rightarrow [color] \\
& \rightarrow (Grid\ color \rightarrow image) \\
& \rightarrow image
\end{aligned}
$$

Note that *draw* is also polymorphic in the type of *image* produced (i.e., *image* is a *type variable*, not a specific type). In specific applications, *image* might be instantiated to a type of values representing images, or to the type of an *IO* action that will draw the result, write it to a file, or perhaps even post it on the web!

4.1 Character-based Pictures of the Mandelbrot Set

It is possible to produce quite attractive pictures of the Mandelbrot set using only simple character output. The first step is to define a palette of characters.

```
charPalette  ::  [Char]
charPalette  =   "     ,.`\"~:;o-!|?/<>X+={^O#%&@8*$"
```

In choosing a value here for *charPalette*, we have made a modest attempt to select characters in a rough progression from light (starting with several spaces) to dark.

Next, we must define the process for rendering a character image: we use *unlines* to append newlines and concatenate the strings in each row of a *Grid Char*, and then *putStr* to display the result:

```
charRender  ::  Grid Char → IO ()
charRender  =   putStr . unlines
```

For example, we can generate Figure 1 using the following:

```
figure1  =  draw points mandelbrot charPalette charRender
            where points  =  grid 79 37 (−2.25, −1.5) (0.75, 1.5)
```

From this starting point, interested readers can begin to explore the Mandelbrot set on their own by varying parameters and generating new images. For example, some images might be enhanced by the use of a different palette; the first two parameters of *grid* could be changed to accommodate a different display or page size; and the last two parameters of *grid* could be changed to focus more closely on particular sections of the Mandelbrot set[2]. We will not explore these possibilities

[2] For example, the regions specified by each of the following pairs of points are worth a closer look: ((−1.15), 0.19) ((−0.75), 0.39), ((−0.19920), 1.01480) ((−0.12954), 1.06707), ((−0.95), 0.23333) ((−0.88333), 0.3), and ((−0.713), 0.49216) ((−0.4082), 0.71429)!

further in this paper, and instead move on to show how different types of fractals, and different methods of rendering can be accommodated by other simple changes.

4.2 Pictures of Julia Sets

Mandelbrot's study of the set that now carries his name was prompted by work that had been done much earlier (and without the aid of a computer) by the French mathematician Gaston Julia (Julia, 1918). Indeed, Mandelbrot's work revived an interest in Julia's work that had been somewhat overlooked, even though it had been awarded the Grand Prix de l'Académie des Sciences at the time it was first published. Today, Julia's name is associated with a collection of fractals known as *Julia Sets*, and, in this section, we will show how the framework presented in earlier sections can be used to draw pictures of Julia Sets like the one shown in Figure 2. In fact, Julia sets are constructed with the same basic machinery that we used to

```
                   ,,,,,,,,,,,,,,,,,,,,,,,,,
               ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
            ,,,,,,,.........................,,,,,,,
          ,,,,,,,,...‘‘"~/*:o~"‘‘‘‘‘‘‘"~~-:"‘‘‘...,,,,,,,,
        ,,,,,,,..‘";$>?$$$$*{$:~"~~|$$$$$$?$$$-"‘..,,,,,,,
       ,,,,,,...‘"$$$$$$$$<?|-o;;o8$$$$$$$$$$$8X$~‘...,,,,,,
     ,,,,,,...‘‘"~$$$$$$$^<?|!--|{$$$$$$$$$$$=?!o:~"‘...,,,,,,
    ,,,,,,..‘"o$>-!$8$$$$$$$$<X#${$$$$$$$$$$$$$$$$$$$"‘..,,,,,,
   ,,,,,,..‘"/$$=<>$$$$$$$$$&$$$$$$$$$$$$$$$$$$$$$$$$$$;‘..,,,,,
   ,,,,,,..‘~$$$$$$$$$$$$$$$$$$$$$$$$$$$$$@@$$$$$$$$$$/!~‘..,,,,,,
  ,,,,,,,...‘~X|$$$$$$$$$$$$$$$$$$$$$$$$$$=$$%$X$$$$$$$|‘...,,,,,
  ,,,,,,,...‘;$$$$$$$$$$$$$$$$$$$$$$$$$$$$<?||||?/<$$$<$‘...,,,,,,
  ,,,,,,,...‘";?$$$$$$$$$$$$$$$$$$$$%0^%$?|!!!!!|?/+$$X"‘..,,,,,,,
  ,,,,,,,...‘‘~$$$$$$$$$$$$$$$$$$$$$$$$X?|!--ooo-$$$/:"‘...,,,,,,,
 ,,,,,,,,,...‘"~-$$$$$$$$$$$$$$$$$$$$$$$$$|!-o;;;::~~"‘‘‘...,,,,,,,
 ,,,,,,,,,...‘‘""~;$$$$$$$$$$$$$$$$$$$$$0$|-o;;:~~""‘‘‘...,,,,,,,,
 ,,,,,,,,,...‘‘‘""~:;o!$$$$$$$$$$$$$$$$$$$$$$!o;:~""‘‘‘...,,,,,,,,,
 ,,,,,,,,,...‘‘‘""~~:;;o-|$0$$$$$$$$$$$$$$$$$$$$$$;~""‘‘...,,,,,,,,,
 ,,,,,,,,...‘‘‘"~~::;;;o-!|$$$$$$$$$$$$$$$$$$$$$$$$-~"‘...,,,,,,,,,
 ,,,,,,,,,...‘":/$$$-ooo--!|?X$$$$$$$$$$$$$$$$$$$$$$$$~‘‘...,,,,,,,
 ,,,,,,,,..‘"X$$+/?|!!!!!|?$%^0%$$$$$$$$$$$$$$$$$$$$?;"‘...,,,,,,,
 ,,,,,,,...‘$<$$$</?||||?<$$$$$$$$$$$$$$$$$$$$$$$$$$$$;‘...,,,,,,,
  ,,,,,,...‘|$$$$$$$X$%$$=$$$$$$$$$$$$$$$$$$$$$$$$$$|X~‘...,,,,,,
  ,,,,,,..‘~!/$$$$$$$$$@@$$$$$$$$$$$$$$$$$$$$$$$$$$$$$~‘..,,,,,,
   ,,,,,..‘;$$$$$$$$$$$$$$$$$$$$$$$$$$&$$$$$$$$$><=$$<"‘..,,,,,,
   ,,,,,,..‘"$$$$$$$$$$$$$$$$$$$${$#X<$$$$$$$8$!->$o"‘..,,,,,,
    ,,,,,,,...‘"~:o!?=$$$$$$$$$$${|--!|?<^$$$$$$$~"‘‘...,,,,,,
      ,,,,,,...‘~$X8$$$$$$$$$$$$8o;;o-|?<$$$$$$$$"‘...,,,,,,
       ,,,,,,,..‘"-$$$?$$$$$$|~~"~:${*$$$$?>$;"‘..,,,,,,,
         ,,,,,,,,...‘‘‘":-~~"‘‘‘‘‘‘‘"~o:*/~"‘‘...,,,,,,,,
           ,,,,,,,.........................,,,,,,,
              ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
                  ,,,,,,,,,,,,,,,,,,,,,,,,,
```

Fig. 2. The Julia set for (0.32,0.043)

investigate the Mandelbrot set. However, instead of fixing the starting point of each sequence that we produce to the origin and varying the first parameter to *next*—as we did for the Mandelbrot Set—we construct sequences for Julia sets by fixing the first parameter, and varying the starting point. This gives us a different way to produce sequences from points, and hence to produce some new fractal images.

$$\begin{aligned} &\mathit{julia} &&:: \quad \mathit{Point} \rightarrow \mathit{Point} \rightarrow [\mathit{Point}] \\ &\mathit{julia}\ c &&= \quad \mathit{iterate}\ (\mathit{next}\ c) \end{aligned}$$

For example, here is the code to produce the picture shown in Figure 2:

$$\begin{aligned} \mathit{figure2} \quad = \quad & \mathit{draw}\ \mathit{points}\ (\mathit{julia}\ (0.32,\ 0.043))\ \mathit{charPalette}\ \mathit{charRender} \\ & \textbf{where}\ \mathit{points} \quad = \quad \mathit{grid}\ 79\ 37\ (-1.5,\ -1.5)\ (1.5,\ 1.5) \end{aligned}$$

Again, we encourage the interested reader to probe more deeply into the structure of Julia sets by playing with different parameter settings. Of course, it is possible to experiment as before with different palettes and with different parameters for *grid*. For Julia Sets, however, there is an additional degree of freedom that can be explored by varying the choice of point that is passed as the first parameter to *julia*.

4.3 Using Colors and High-resolution Graphics

Another way to render fractal images is to display them using high-resolution graphics, with one colored pixel for each point in the input grid. We can modify our code to draw images like this given only a few simple primitives: a palette of colors; a way to create a canvas for drawing; and a way to set the color of individual pixels.

$$\begin{aligned} &\mathit{rgbPalette} &&:: \quad [\mathit{RGB}] \\ &\mathit{graphicsWindow} &&:: \quad \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{IO}\ \ \mathit{Window} \\ &\mathit{setPixel} &&:: \quad \mathit{Window} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{RGB} \rightarrow \mathit{IO}\ () \end{aligned}$$

It is easy to implement these functions using the facilities provided by the Hugs graphics library (Reid, 2001). The types above reflect this heritage in their use of the *RGB* type for colors and the *Window* type for a graphics window. Further details of our implementation, however, are not particularly interesting and hence will not be shown here. It should be quite simple to reimplement the same functionality using other graphical toolkits or libraries.

With these pieces in hand, we can define a simple rendering function for grids of *RGB* values. Apart from drawing the image, much of the following code has to do with creating a window, waiting for the user to hit a key when they have seen the result, and then closing the window:

$$\begin{aligned} &\mathit{rgbRender} &&:: \quad \mathit{Grid}\ \mathit{RGB} \rightarrow \mathit{IO}\ () \\ &\mathit{rgbRender}\ g &&= \quad \textbf{do}\ w \leftarrow \mathit{graphicsWindow}\ (\mathit{length}\ (\mathit{head}\ g))\ (\mathit{length}\ g) \\ & && \qquad\quad \mathit{sequence\_}\ [\mathit{setPixel}\ w\ x\ y\ c \mid (\mathit{row},\ y) \leftarrow \mathit{zip}\ g\ [0..], \\ & && \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (c,\ x) \leftarrow \mathit{zip}\ \mathit{row}\ [0..]] \\ & && \qquad\quad \mathit{getKey}\ w \\ & && \qquad\quad \mathit{closeWindow}\ w \end{aligned}$$

Graphical versions of the Mandelbrot and Julia Set images that we saw in previous Figures are shown in Figure 3. Apart from the change of palette and renderer, these

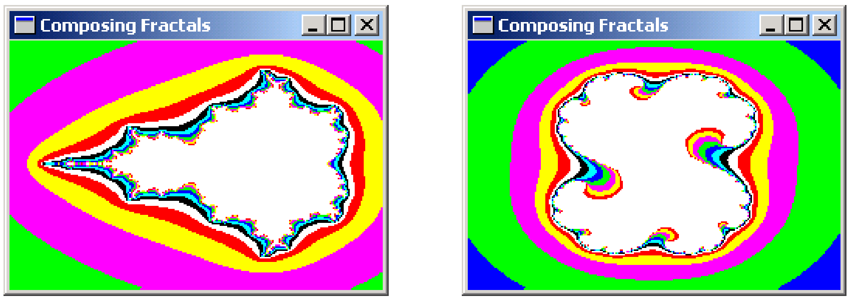


Fig. 3. Graphical Displays of Mandelbrot and Julia Sets

images use the same parameters as in the original figures but with a finer resolution grid of 240 by 160 pixels.

```
figure3left   =  draw points mandelbrot rgbPalette rgbRender
                 where points  =  grid 240 160 (−2.25, −1.5) (0.75, 1.5)

figure3right  =  draw points (julia (0.32, 0.043)) rgbPalette rgbRender
                 where points  =  grid 240 160 (−1.5, −1.5) (1.5, 1.5)
```

Once again, there are plenty of opportunities for an interested reader to experiment with other choices of parameters!

5 Closing Thoughts

The programs described in this paper demonstrate how functional languages, like Haskell, can support an appealing, high-level approach to program construction that lets independent aspects of program behavior be expressed in independent sections of program text. We have shown that the resulting code is easy to adapt and modify so that it can be used in a variety of different settings. Of course, it is possible to program in a compositional manner in other languages, but the style seems particularly natural in a functional language, where features like polymorphism, higher-order functions, laziness, and lightweight syntax can each contribute, quietly, to elegant and flexible programming solutions.

Several authors have demonstrated the role that functional programming languages can play in graphics, particularly in describing images like fractals that have a rich mathematical structure (Henderson, 1982; Hudak, 2000; Elliott, 2003). For those readers with an interest in learning more about the mathematics of fractals—or even just in taking a look at many beautiful fractal images—we recommend the book by Peitgen and Richter (1988). There are, of course, several other books, and numerous web sites with further information and images.

Acknowledgments

Thanks to Ralf Hinze, Levent Erkök, Melanie Jones, Philip Quitslund, Tom Harke, and five anonymous referees whose comments helped to improve the content and presentation in this paper.

References

Bird, R. (1998) *Introduction to Functional Programming (2nd Edition).* Prentice Hall PTR.

Elliott, C. (2003) Functional images. Gibbons, J. and de Moor, O. (eds), *The Fun of Programming.* Palgrave Macmillan.

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

Hudak, P. (2000) *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press.

Julia, G. (1918) Mémoire sur l'itération des fonctions rationnelles. *Journal de Math. Pure et Appl.* **8**:47–245.

Mandelbrot, B. B. (1975) *Les objets fractals: forme, hasard et dimension.* Flammarion.

Mandelbrot, B. B. (1988) *The Fractal Geometry of Nature.* W.H. Freeman & Co.

Peitgen, H.-O. and Richter, P. H. (1988) *The Beauty of Fractals: Images of Complex Dynamical Systems.* Springer Verlag.

Peyton Jones, S. (ed). (2003) *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press. See also `http://www.haskell.org/definition/`.

Reid, A. (2001) *The Hugs Graphics Library (Version 2.0).* available from `http://www.haskell.org/graphics`.

Thompson, S. (1999) *Haskell: The Craft of Functional Programming (2nd Edition).* Addison-Wesley.