A Case Study in Haskell
Implementing a Fractal Generator, Frac

Prepared for COMP6411,
and presented to Dr. Constantinides

Simon J. Symeonidis
5887887

February 17, 2015

Contents

1 Introduction **1**

2 Language Exploration **1**
2.1 Types . . . 1
2.2 Recursion . . . 3
2.3 Visibility & Signatures . . . 4
2.3.1 Generics . . . 5
2.3.2 Higher Order Functions . . . 5
2.4 Defining your own Infix Operators . . . 6

3 Problem Dissection **7**
3.1 Writing the Image Library . . . 7
3.2 Julia Sets and Fractal Implementation . . . 7
3.3 Image Output . . . 8

4 Code **15**

Listings

1 Simple List Operations . . . 1
2 Strings are Character Lists (Don't believe what they told you) . . . 2
3 Cases using argument values . . . 3
4 Factorial in Haskell . . . 3
5 Factorial Output . . . 3
6 Exposing Functions in a Module . . . 4
7 Example Module with Function . . . 4
8 Generic toStr . . . 5
9 Generic Output . . . 5
10 Our Own Filter . . . 6
11 99 Bottles of Beer in Haskell . . . 6
12 Main Entry Point . . . 15
13 PPM Filters . . . 15
14 PPM Patterns . . . 16
15 Fractal Implementation . . . 17
16 Simple Fractals . . . 17
17 PPM Package . . . 20

::Section 1::

Introduction

This is my coursework for COMP6411, on the Haskell programming language. Inside this part we discus the different powers of the programming language, the flexibilities, and other shortcommings.

The application in question is a fractal generator. In order to provide a solution to this problem, we need to decide on two things:

1. Decide on what fractal algorithm we will choose
2. Decide on how we will display the outcome

This clearly separates this case study into two parts. One part is to define a library that we can call in order to generate images. The other part is to implement the actual fractal algorithm, and auxiliary functions.

::Section 2::

Language Exploration

In this section we show aspects of the programming language. We show the typing system and signatures, functions, higher order functions, and recursive functions. We also show how some basic *Haskell* applications or modules are written and used.

2.1 Types

We discus some basic *Haskell* types in this section. The operations listed were what were deemed useful for the case study. There exists many other possible operations that a user of the language can use, and it is left for the user for exploration.

Lists are used by ensaring elements between square brackets. For example, [1, 10, 2, 4] is a valid list. Lists are required to contain elements of the same type however. For example [1, 2.0, "cat"] would raise compilation issues. We are provided with the *head* and *tail* operations in order to manipulate lists. We are also given some very useful operations called *take* and *drop* which returns the first n specified elements, or returns the tail after n elements respectively. Mapping functionality exists, so applying lambdas on lists is also possible. Another useful operator is the *!!* infix operator, which returns us an element at a given index. Some sample output is provided in Listing 1. There exists a *cons* operator, and in *Haskell* it is the colon.

```
Prelude> [1,2,3,4]
[1,2,3,4]
Prelude> [1,2,3,"cat"]

<interactive>:11:2:
    No instance for (Num [Char]) arising from the literal '1'
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: 1
    In the expression: [1, 2, 3, "cat"]
    In an equation for 'it': it = [1, 2, 3, ....]

Prelude> head [1,2,3,4]
1
Prelude> tail [1,2,3,4]
[2,3,4]
Prelude> take 2 [1,2,3,4]
[1,2]
```

```
Prelude> drop 1 [1,2,3,4]
[2,3,4]
Prelude> [1,2,3,4] !! 0
1
Prelude> [1,2,3,4] !! 2
3
Prelude> 12 : [1, 2, 3]
[12,1,2,3]
Prelude> "mommy cat" : ["kitten", "kitten", "kitten", "kitten"]
["mommy cat","kitten","kitten","kitten","kitten"]
```

Listing 1: Simple List Operations

Another interesting feature in the lists of *Haskell* is that we are able to define ranges, and evaluate them lazily. We use two periods between two numbers in order to achieve this. The following are valid ranges: [1..100], [0.5..0.9]. If we omitt the rightmost number, then we tell *Haskell* that it is a list that goes on to infinity. This is legal due to the lazy nature.

Another very useful aspect of the language is that we can use *list comprehensions*. List comprehensions and lazy evaluation add to the power of the language dramatically.

Strings can be specified by using the quotation marks. For example “My kitten is cute” is a valid string. Strings however are lists of characters. They have been aliased as the type *String*. So in fact, we can say $String = [Char]$. It follows that any operation that we can perform on a list as previously stated, is possible to be used in Strings as well. For example, the head of the string “hello” would return the character ‘h’, and the tail would return “ello”. In this respect, we can also map functions onto strings. Listing 2.

```
Prelude> :t "hello"
"hello" :: [Char]
Prelude> ["hello", "there"]
["hello","there"]
Prelude> :t ["hello", "there"]
["hello", "there"] :: [[Char]]
Prelude> head "hello"
'h'
Prelude> tail "hello"
"ello"
Prelude> Data.Char.toUpper 'a'
'A'
Prelude> map Data.Char.toUpper "hello there"
"HELLO THERE"
Prelude> take 4 "hello"
"hell"
Prelude> 's' : take 4 "hello"
"shell"
```

Listing 2: Strings are Character Lists (Don’t believe what they told you)

Others We also have other primitives such as Booleans, Integers, Doubles, and tuples in our disposal. Tuples are defined by using parentheses, and separating its types with commas: (String, Int, Double) is a valid tuple.

2.2 Recursion

Recursion is possible in *Haskell*. Similarly to *Prolog* we are given to separate the base cases from the body of the function. The recursive cases remain in the body of said function. Therefore, the base cases are evaluated in the order they are appear in. For example consider the code in Listing 3.

```haskell

module MyTest where

mycase 1 = 10
mycase 2 = 20
mycase 3 = 30
```

Listing 3: Cases using argument values

If we executed *mycase 1* we would return the value '10'. If we executed *mycase 2* we would return the value '20'. If we executed this function with the argument value '4', then we would get an error.

Now provided the mechanism on how to define base cases, let us take a look how a factorial implementation would look like in *Haskell*. Listing 4 shows us this.

```haskell
module Factorial(myfactorial) where

myfactorial :: Int -> Int
myfactorial n =
  case n > -1 of
    True  -> factorialback n
    False -> error "Please supply positive integers"

factorialback :: Int -> Int
factorialback 0 = 1
factorialback n = n * factorialback (n - 1)
```

Listing 4: Factorial in Haskell

Listing 5 shows us some example usage.

```
*Factorial> myfactorial 3
6
*Factorial> myfactorial 10
3628800
*Factorial> myfactorial 14
87178291200
*Factorial> myfactorial (-3)
*** Exception: Please supply positive integers
```

Listing 5: Factorial Output

One might notice that there is no safeguard for giving negative integers. This is prevented actually on compile time.

2.3 Visibility & Signatures

In comparison to other languages such as *C, C++, OCaml,* Haskell does not define its interfaces to its modules in a separate file - there is no separation between specification and implementation. In order to achieve visibility, we need to specify the function names which we want to expose between brackets after the module name. We can see such an example in Listing 6.

Haskell is a strong typed language. This adds to the compiler's task at seeing if the user of the language is trying to do something erroneous. As such, *Haskell* does not support casting, or other features that might cause problems during runtime. The idea is to catch as many bugs as possible during compile time, using the type system.

```haskell
-- @author Simon Symeonidis
module MyModule(toStrList) where

...
```

Listing 6: Exposing Functions in a Module

Signatures exist in Haskell. They can be thought of as prototype functions in *C/C++*. A Haskell signature is similar to those of *OCaml*. We need to specify a name, then add a double colon, and separate it with its type signature which contains the return type as the last type. Here is an example:

$$functionName :: \langle signature \rangle$$

We can see a more implemented example in Listing 7. We can observe the function name *toStrList*, on its right the separator '::', and then the type signature *[Int] -⟩ [String]*. Recall that we said that the last type is the return type. Therefore this makes us understand that all the types right before the rightmost are the inputed parameters. So in the case of *toStrList*, we understand that this function takes in a List of integers, and returns a list of strings.

```haskell
-- @author Simon Symeonidis
module MyModule(toStrList) where

-- | Example where we convert an integer list to a string list. The signature
-- is on the first line bellow. The second line below it uses the map function
-- performing `show` on each element which converts it to a String. We return
-- a list of strings, as specified by the signature.
toStrList :: [Int] -> [String]
toStrList lst = map show lst

-- Example usage:
-- [psyomn@aeolus misc 0]$ ghci MyModule.hs
-- GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
-- Loading package ghc-prim ... linking ... done.
-- Loading package integer-gmp ... linking ... done.
-- Loading package base ... linking ... done.
-- [1 of 1] Compiling MyModule         ( MyModule.hs, interpreted )
-- Ok, modules loaded: MyModule.
-- *MyModule> toStrList [1,2,3,4,12313,123123123,231]
-- ["1","2","3","4","12313","123123123","231"]
```

Listing 7: Example Module with Function

2.3.1 Generics

Generics can be thought of like templates in *C/C++ or Java.* This means that the possibility of reusing a function with many types is possible. Let us revisit the example where we were converting integers into lists (Listing 7). This is a nice function, but it only works for *Integer* input. What if we wanted to use it for various data types? This is where the use of generics are deemed useful. A generic is defined by a lowercase character. So if we want a list of 'things', whatever they may be, we denote that this way: $[a]$. Extending on the previous example, we first need to change the type signature. Since we want any type and not only Integers, we need to replace the parameter. So the signature now becomes: $[a] \to [String]$.

However there is one last thing that we need to take care about our generic type 'a'. If we wish for the possibility of converting each element 'a' into a string, we need to impose the type constraint *Show.* This requires that any type we pass to this new function should have behavior defined for 'show' (which is similar to toString functions in other programming languages). The signature now becomes: $(Show\ a)\ =\rangle [a] \to [String]$.

As you might have inferred, constraints are imposed by writing them at the left-most of the type signature. If more constraints were to be added, you would separate them by commas. So for example if we also wanted the generic type 'a' to be constrained by 'Ord' (if something is order-able), we would change the constraint to: $(Show\ a,\ Ord\ a)$. Listing 8 shows the generic implementation of the list to string function, and 9 shows the output when providing different lists of different element types.

```haskell
module MyModuleGeneric (genToStr) where

genToStr :: (Show a) => [a] -> [String]
genToStr things = map show things
```

Listing 8: Generic toStr

```
*MyModuleGeneric> genToStr [1,2,3,4]
["1","2","3","4"]
*MyModuleGeneric> genToStr [1.0, 2.0, 3.0]
["1.0","2.0","3.0"]
*MyModuleGeneric> genToStr[True, False, True, True]
["True","False","True","True"]
```

Listing 9: Generic Output

2.3.2 Higher Order Functions

Higher order functions are supported in *Haskell.* One thing to note is how to express them inside the function you are defining, that will use that function. The rest is not too hard to get working. For the sake of this example we will be implementing our own filter. (Recall that a filter tests the list's elements against a predicate, and are added to the new list if they satisfy said predicate).

First we need to define the signature. We know there are two inputs: the predicate, and the list to test it against. We expect a list, empty or non-empty after the operation has completed. We can first piece together the following: $function \to [a] \to [a]$. That is we expect after passing the function and a list of elements type 'a', to return a list of type element 'a'.

Next we need to define the function in more detail. To specify a higher order function we need to encase the signature in parentheses. Our predicate gets an element 'a', and returns a 'Boolean'. Therefore the predicate inside the function we are trying to define looks like $(a \to Bool)$.

Putting the signature together we have the following: $myfilter :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$. The implementation uses recursion in order to build the list, depending on whether the predicate is satisfied or not. Listing 10 shows the final implementation of the function.

```haskell
module MyFilter(myfilter) where

myfilter :: (a -> Bool) -> [a] -> [a]
myfilter _ []     = []
myfilter f (x:xs) =
  case f x of
    True  -> x : myfilter f xs
    False -> myfilter f xs
```

Listing 10: Our Own Filter

You may notice the underscore in the base case of this function. That is an anonymous variable. Since we do not care about the predicate in the base case we are free to omitt it.

2.4 Defining your own Infix Operators

You can also define your own infix operators. This is a useful feature in the programming language. Here is an example where we used this in order to implement the '99 bottles of beer' problem. Notice the backtics on 'bottles' in Listing 11.

```haskell
{- Using the infix operator, trying to make more concise code.
   @author Simon Symeonidis -}

module Main(main) where

import Text.Printf

bottlesPlurals :: Int -> String
bottlesPlurals x = case x == 1 of
    True  -> "bottle"
    False -> "bottles"

bottles :: Int -> String -> String
0 `bottles` str = "No more " ++ (bottlesPlurals 0) ++ str
x `bottles` str = (show x) ++ " " ++ (bottlesPlurals x) ++ str

makeSong :: Int -> String
makeSong 0 = ""
makeSong x =
  x `bottles` " of beer on the wall, " ++
  x `bottles` "  of beer. " ++ "Take one down and pass it around, " ++
  (x - 1) `bottles` " of beer on the wall. " ++ makeSong (x - 1)

main = do
  printf $ makeSong 99
```

Listing 11: 99 Bottles of Beer in Haskell

::Section 3::

Problem Dissection

For this case study, I decided to write up a fractal generator. This required two steps. Writing a small library for outputing images, and writing the fractal mathematical formulas.

3.1 Writing the Image Library

I was looking for a specification of an image format that would not be too hard to implement due to the time constraint I had for this project. Therefore the ideal format for my application would be something that could be represented as text. Apart from being really easy to implement, it would be easy to debug as well, since the output would be text files.

Searching on the Internet I finally stumbled upon the *PPM* format for images. There are two versions of the format one may implement. This is read by the image viewer on the file headers. If you specify P6, then it expects to read the image data as raw bits. However if you specify P3, you can enter the image data by 3-tuples of pixels (where the tuple contains the luminoscity of the Red, Green, and Blue colors). More information on the file format can be read about here: `http://netpbm.sourceforge.net/doc/ppm.html`.

The best approach for this was to define a data type *Pixel* which contained these three colors. Then we could define auxiliary functions to modify the information, or read it. Such behavior is found in Listing 17. Notice first the two types: Pixel, and PPMImage. The pixel defines another 3 elements which will be placeholders for the RGB colors, and PPMImage, is the whole image - the header information, dimensions and image data included. Using these two types, we build the foundations for completing the rest of the application.

We can notice some of the auxiliary functions *redOf, blueOf, greenOf.* Those are used in order to extract the luminoscity of the pixels. Notice how the anonymous variables are used in order to only get the data that we want for each.

The next important functions are *makeRow, makeImage, and makeImageData.* These functions help us create the image in question. The *makeRow* function creates a row of pixels. This function is used 'n' times in the back-end of the *makeImage* function in order to give height to the image. We have yet more auxiliary functions such as *makeWhiteImage, makeGreenImage, makeBlackImage, etc* which provide us shorthands to create images of x, y dimensions. The *makeBlackImage* function is used later in order to provide a background to the fractal.

Last but not least are the string conversion functions that return the image as a string. This is important because as previously stated, we wish to export the image into a text format that is viewable as a picture by an image viewer.

First we need to take care of the finely grained issue - converting a single pixel into a tuple of three numbers as a string. This is observed in the *pixelString* function in Listing 17. Then we define a recursive function that goes through a whole row in an image, and converts it into a string - namely *outputRow.* Finally we define one last function to take care of output for a whole image, *imageString, imageStringBackend.* This concludes the requirements of having a library that is able to generate our images.

3.2 Julia Sets and Fractal Implementation

The next important listing to look at is 16. We wish to investigate Julia sets using the quadratic polynomial below. The quadratic polynomial was obtained from here: `http://en.wikipedia.org/wiki/Mandelbrot_set`. The 'z' values are always replaced by the previous value that was obtained calculating the function with constant 'c'. Constant 'c' is a arbitrary imaginary number that is supplied.

$$z_{n+1} = z_n^2 + c$$

Now there are two things that we need to consider: the ranges that the set is defined inside, and the range of the image dimensions. That is, we need to super impose the dimensions of the cartesian plane where the fractal is defined, on an array representing an image. Therefore, for each array index that we have in our image, we stretch the range of the cartesian plane over it. This means that each index adds a small fraction to the dimension. If for example we had an image of size 2000 pixels wide, and our cartesian plane was from the range -2 to 2, then for each pixel, we need to add a small amount that is calculated in the 'grain' function in Listing 16. This function translates to the following:

$$grain = \frac{abs(minv) + abs(maxv)}{distance_{minmax}}$$

A list comprehension is used in order to poppulate the array with these values. The list comprehension can be spotted inside the function *makePlane*.

Another aspect to take into consideration is that we need to calculate values using imaginary numbers. To do this, we import the complex numbers in Haskell. We can define imaginary numbers like this: *1.0 :+ 2.0*, where the left part is the real part, and the right part is the imaginary part. Operators are overloaded in order to perform calculations if needed.

The plane is poppulated with these imaginary values. The real parts are affected from left to right, by the grains as stated (so for each succeeding pixel from left to right, we add 'n' times this grain). The imaginary parts are affected form top to bottom (using the +y to -y values for the grain formula).

Finally we need to apply the polynomial function z recursively on a pixel's imaginary values. The formula is repeatedly called, using the previous output as its input. Each time, we test the Euclidean distance from the center of the plane, to the new coordinate. The Euclidean distance uses the raw magnitudes of the complex numbers for the 'x' and 'y' inputs.

$$euclidean = \sqrt{abs(magnitude_{real})^2 + abs(magnitude_{imaginary})^2}$$

We keep track at how many times we applied the z function until we got a value more than 2, when calculating its Euclidean distance. If it's greater than 2, then it means that it has escaped the plane of interest and that that point does not belong to the Julia set. However, if it takes a considerable amount of iterations in order to escape this boundary, we note this point as belonging to the Julia set. The points belonging to the Julia set are what make up the visible part of the fractal on pictures. Also, the intensity of that particular point's color is determined by the amount of iterations required in order for it to escape (or remain) in the plane.

3.3 Image Output

Here are some fractals, generated by the program, providing different constants 'c'. Note that the colors were edited after the output, for facilitating printing.

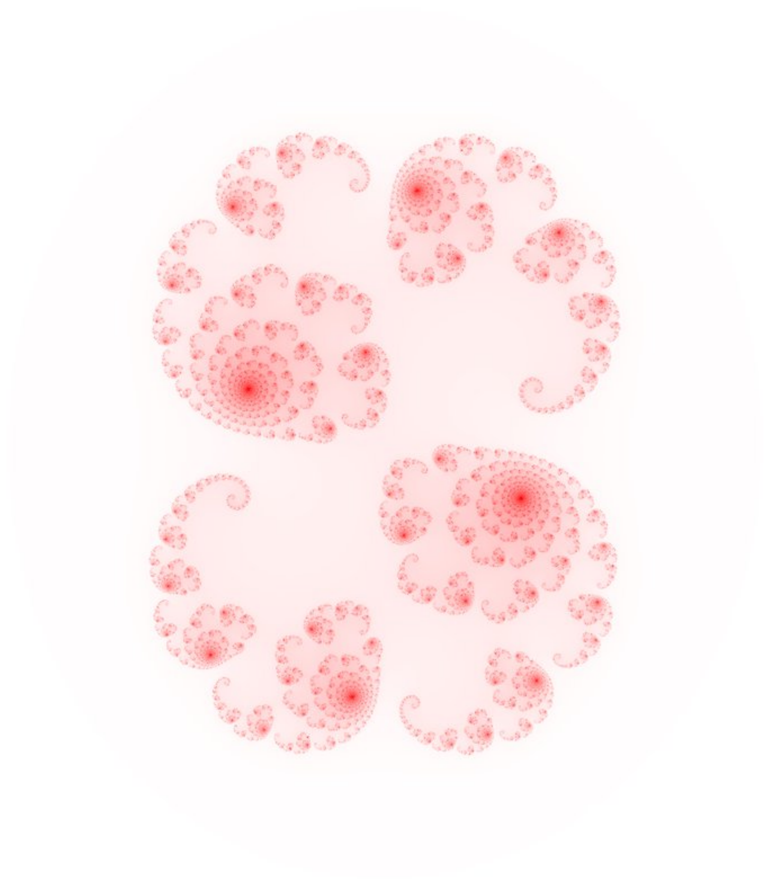

Figure 1: Fractal 1

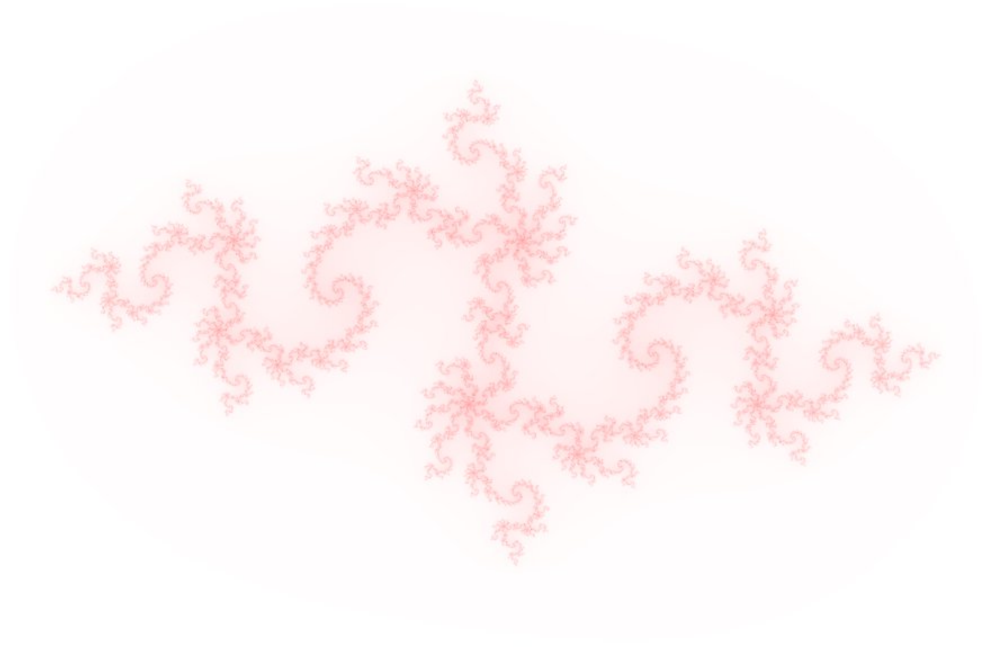

Figure 2: Fractal 2

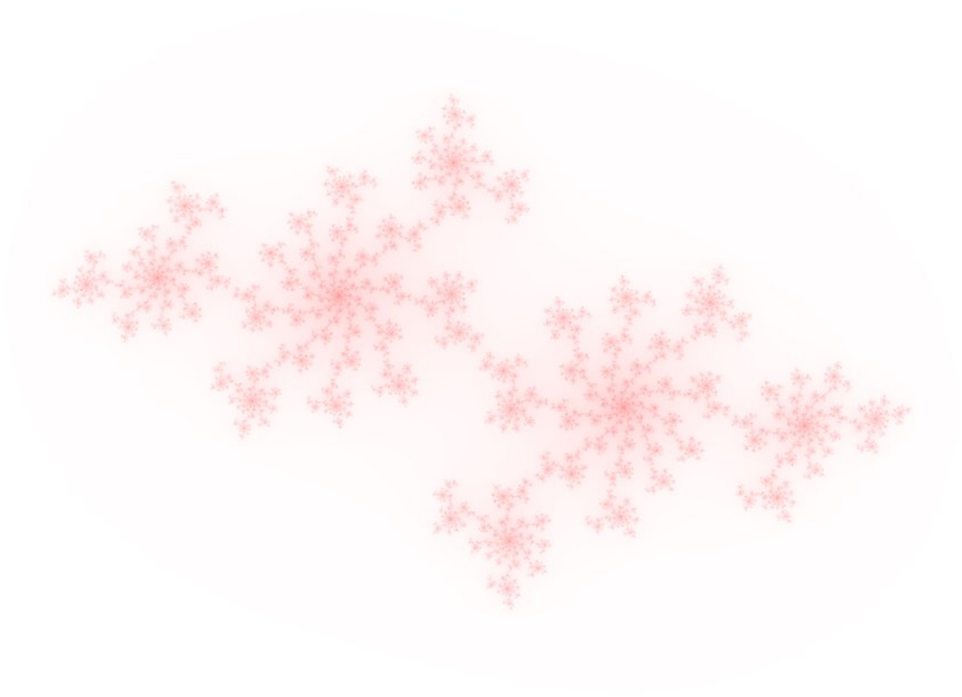

Figure 3: Fractal 3

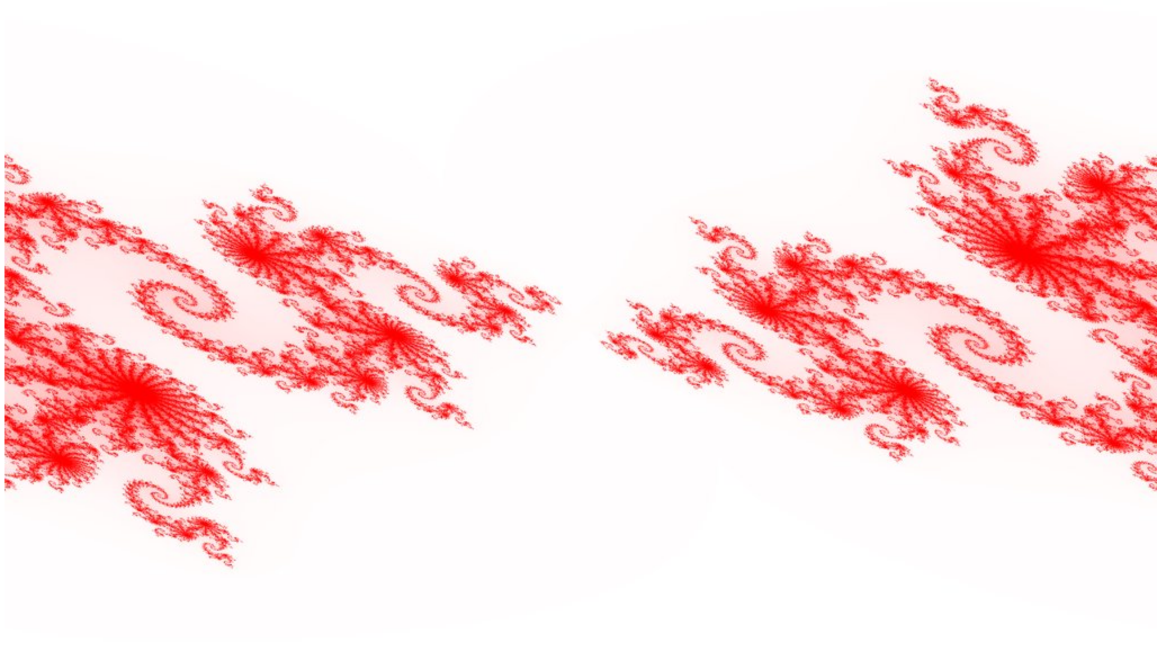

Figure 4: Fractal 4

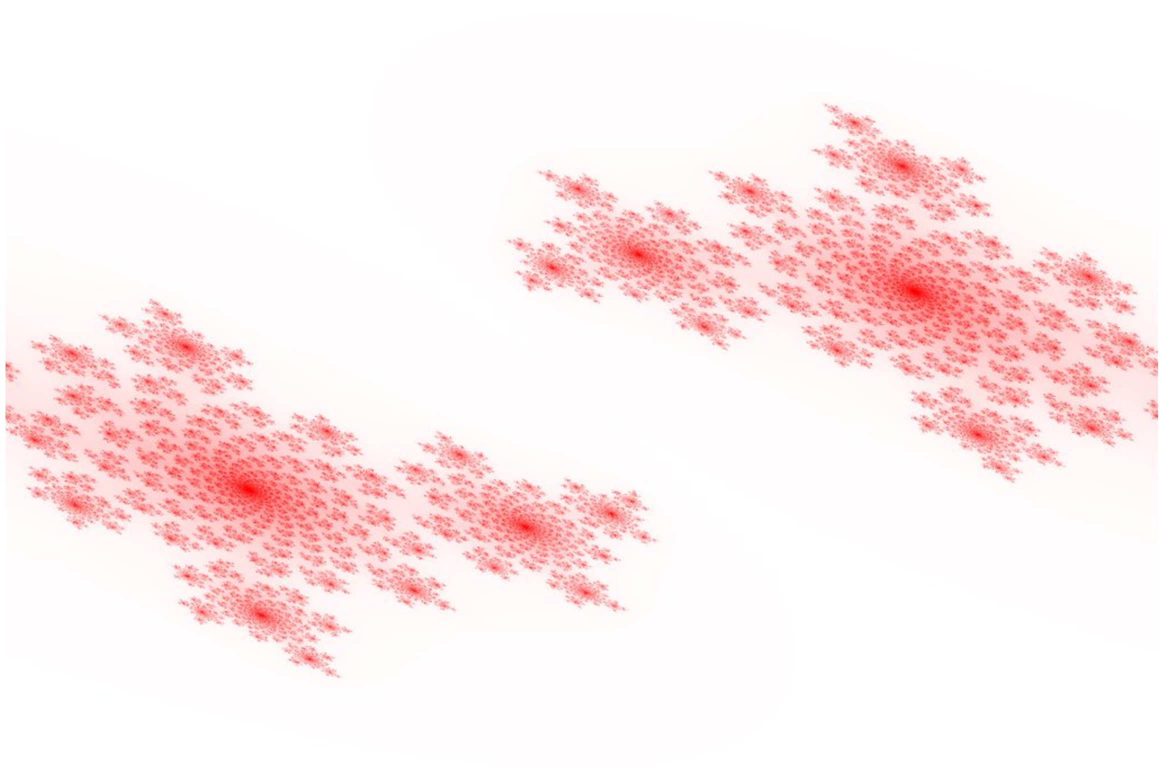

Figure 5: Fractal 5

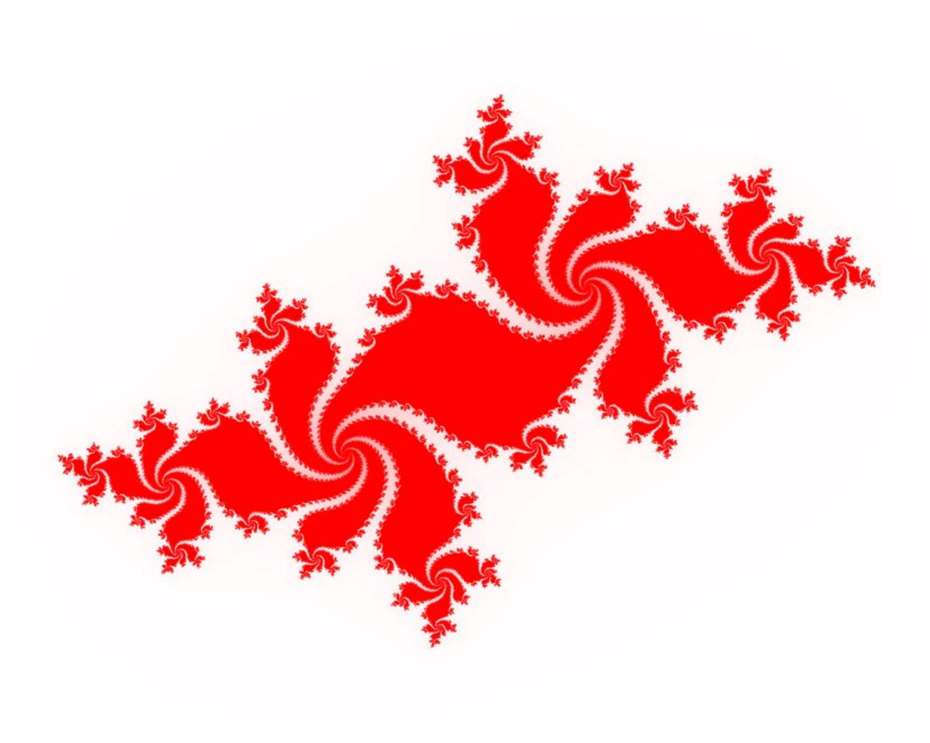

Figure 6: Fractal 6

Code

In this section we list all the source code that was used in order to complete the fractal generator.

```haskell
-- @author Simon Symeonidis
--   Trying to make a haskell output some pretty fractal graphics
module Main(main) where

import Text.Printf

import Ppm
import Ppm
import Ppm.Patterns
import Ppm.Filters

import Fractals.Simple

pixelBool b =
  case b == True of
    True  -> (Pixel 255 255 255)
    False -> (Pixel   0   0   0)

buildImage      [] = []
buildImage  (x:xs) = map pixelBool x : buildImage xs

maxPlaneX = 2000
maxPlaneY = 2000
plane     = makePlane (-2) 2 maxPlaneX (-2) 2 maxPlaneY
slplane   = slice (2001) plane
-- ^ +1 because each row has 0.0 as a point

boolPlane  = calculateBool    julia slplane
colorPlane = calculateColored julia slplane

imgDataBool = buildImage boolPlane

main = do
  let image = makeRedImage 2001 2000
  let out   = setData image colorPlane
  printf $ outputImage out
```

Listing 12: Main Entry Point

```haskell
{- Fun, trivial filters to manipulate the ppm files.
   @author Simon Symeonidis -}
module Ppm.Filters (
Direction
, revRow
, shadow
) where

import Ppm
```

```haskell

data Direction = Left | Right | Up | Down

shadow :: [[Pixel Int]] -> [[Pixel Int]]
shadow imgdat = shadowBack imgdat 1

shadowBack :: [[Pixel Int]] -> Int -> [[Pixel Int]]
shadowBack []         i = []
shadowBack (row:rows) i = (shadowRow row i) : shadowBack rows i

shadowRow :: [Pixel Int] -> Int -> [Pixel Int]
shadowRow [] _ = []
shadowRow (p:ps) ix = safeDecreasePixel ix p : shadowRow ps (ix + 1)

safeDecrease :: Int -> Int
safeDecrease x = safeDecreaseAmt x 1

safeDecreaseAmt :: Int -> Int -> Int
safeDecreaseAmt x y = case x - y <= 0 of
                    True  -> 0
                    False -> x - y

safeDecreasePixel :: Int -> Pixel Int -> Pixel Int
safeDecreasePixel 0  pix = pix
safeDecreasePixel ix pix =
  makePixel (safeDecreaseAmt (redOf   pix) ix)
            (safeDecreaseAmt (blueOf  pix) ix)
            (safeDecreaseAmt (greenOf pix) ix)

revRow :: [[Pixel Int]] -> [[Pixel Int]]
revRow imgdat = revRowBack imgdat 0

revRowBack :: [[Pixel Int]] -> Int -> [[Pixel Int]]
revRowBack []         _ = []
revRowBack (row:rows) x =
  case x `mod` 2 == 0 of
    True  -> reverse row : revRowBack rows (x + 1)
    False -> row : revRowBack rows (x + 1)
```

Listing 13: PPM Filters

```haskell
{- Small collections of patterns to make images with.
   @author Simon Symeonidis -}
module Ppm.Patterns(
columated
, checkers
, gradient
) where

import Ppm

{- Make a gradient with blue color-}
gradient :: Int -> Int -> [[Pixel Int]]
gradient _ 0 = []
gradient x y = gradientRow x x : gradient x (y - 1)
```

```

{-
- Make a pixel row
- @param x are pixels left to generate
- @param l is the total row length
-}
gradientRow :: Int -> Int -> [Pixel Int]
gradientRow 0 _ = []
gradientRow x l =
  makePixel 0 (x `mod` 255) 0 : gradientRow (x - 1) l

{- Create Checkers, which are 1x1 in width and height -}
checkers :: Int -> Int -> [[Pixel Int]]
checkers _ 0 = []
checkers x y = checkerRow x y : checkers x (y - 1)

{- We just use x, and y is used for checker effect -}
checkerRow 0 _ = []
checkerRow x y =
  case odd (x + y) of
    True  -> makePixel 15 0 0 : checkerRow (x - 1) y
    False -> makePixel 0 15 0 : checkerRow (x - 1) y

{- Column like pattern -}
columated :: Int -> Int -> [[Pixel Int]]
columated _ 0 = []
columated x y =
  columatedRow x : columated x (y - 1)

columatedRow :: Int -> [Pixel Int]
columatedRow 0 = []
columatedRow x =
  case odd x of
    True  -> makePixel 15 0 0 : columatedRow (x - 1)
    False -> makePixel 0 15 0 : columatedRow (x - 1)
```

Listing 14: PPM Patterns

```

module Fractals (
fibonacciWord
) where

fibonacciWord :: Floating a => a -> a
fibonacciWord phi = 3 * (log phi) / (log ( (3 + sqrt 13) / 2))

-- mandelBrot :: Floating a => a -> a
```

Listing 15: Fractal Implementation

```
{-
  This contains some fractals that may be used in order to generate different
```

```haskell
  information, that can then be pipped to some other image library. This is
  here purely for math stuff.

  @author Simon Symeonidis
-}
module Fractals.Simple (
julia
, makePlane
, slice
, calculate
, calculateColored
, calculateBool) where

import Data.Complex
import Ppm

type CxDouble = Complex Double

maxIterations :: Int
maxIterations = 255

-- |The function allows setting a C of whatever value, but for the sake of
--   simplicity, we're going to use 0
julia :: CxDouble -> CxDouble
-- julia z = z * z + ( (-0.835) :+ (-0.232) )
-- julia z = z * z + ( (-0.4) :+ (-0.6) )
-- julia z = z * z + ( (-0.8) :+ (0.156) )
-- julia z = z * z + ( (-0.70176) :+ (-0.3842) )
-- julia z = z * z + ( (-0.835) :+ (-0.2321) )
julia z = z * z + ( (-0.51) :+ (0.5315) )


-- | Check if the given point, after two iterations of a fractal function will
--   be greater than the radius 2 (there was a theory somewhere that I read
--   that generally if two iterations are greater than 2, then the point will
--   go to infinity).
toInfinity :: (CxDouble -> CxDouble) -> CxDouble -> Bool
toInfinity f point = 2 < (euclidean $ f $ f point)

euclidean :: CxDouble -> Double
euclidean cmp =
  sqrt $ (realPart cmp) ** 2 + (imagPart cmp) ** 2

-- | Create a cartesian plane composed by xs and ys
makePlane x1 x2 xpix y1 y2 ypix =
  [y :+ x | x <- (makeRange x1 x2 xpix), y <- (makeRange y1 y2 ypix)]

-- | The range that we're interested in (for example -2 -> 2).
makeRange :: Double -> Double -> Double -> [Double]
makeRange min max dist =
  reverse $ makeRangeBack min max (grain min max dist) dist

-- | Slice an array to smaller arrays of 'n' size
slice :: Int -> [a] -> [[a]]
slice _  []  = []
slice n arr = take n arr : slice n (drop n arr)

-- | Create a distribution of values in the given array
--   dist is the distribution (how many elements)
--   min is the minimum
--   max is the maximum
makeRangeBack :: Double -> Double -> Double -> Double -> [Double]
makeRangeBack _    _    _  (-1) = []
makeRangeBack min max step curr =
  min + step * curr : makeRangeBack min max step (curr - 1)

-- | Check out how much each step should be in magnitude given two cartesian
--   points, and the length of the actual picture that portrays it.
grain x y d = ((abs x) + (abs y)) / d

-- | Calculating each coordinate, and spitting out the ensuing plane
calculate :: (CxDouble -> CxDouble) -> [[CxDouble]] -> [[CxDouble]]
calculate _ []     = []
calculate f (x:xs) = map f x : calculate f xs

-- | This is to be used if we're just going to make something that displays
--   black and white. I mainly wrote and am using this in order to debug while
--   keeping fingers crossed that I undestood all the theory / articles I read
--   about fractals.
calculateBool :: (CxDouble -> CxDouble) -> [[CxDouble]] -> [[Bool]]
calculateBool _ []     = []
calculateBool f (x:xs) = map (toInfinity f) x : calculateBool f xs

-- | Make a colored fractal
calculateColored :: (CxDouble -> CxDouble) -> [[CxDouble]] -> [[Pixel Int]]
calculateColored _     [] = []
calculateColored f (x:xs) =
  map (funcIterationsPixel f) x : calculateColored f xs

funcIterationsPixel :: (CxDouble -> CxDouble) -> CxDouble -> Pixel Int
funcIterationsPixel f ipoint =
  makePixel 0 (funcIterations f ipoint) (funcIterations f ipoint)

funcIterations :: (CxDouble -> CxDouble) -> CxDouble -> Int
funcIterations f ipoint = funcIterationsBack f ipoint 0

-- | Responsible to actually calculate how many iterations until the point
--   reaches distance > 2
--   f      is the function to apply
--   ipoint is the imaginary point
--   acc    is the accumulator
funcIterationsBack f ipoint acc =
  case euclidean(f ipoint) < 2 && acc < maxIterations of
    True  -> funcIterationsBack f (f ipoint) (acc + 1)
    False -> acc

-- | return a pixel color depending on the magnitude of displacement
colorByMagnitude :: Double -> Pixel Int
colorByMagnitude x
  | x < 0.2 = makePixel 255   0   0
  | x < 0.3 = makePixel   0 255   0
  | x < 0.4 = makePixel 255 255 255
  | x < 0.5 = makePixel 255 255 255
  | x < 0.6 = makePixel 190 190 190
  | x < 0.7 = makePixel 200 200 230
  | x < 1.0 = makePixel 200 230 200
  | x < 1.5 = makePixel 230 200 200
  | x < 2   = makePixel 200 200 200
```

```
    | x < 10  = makePixel  30   0   0
    | x < 20  = makePixel 100   0   0
    | x < 30  = makePixel   0 100   0
    | x < 50  = makePixel   0   0 100
    | x < 60  = makePixel 100 100   0
    | x > 59  = makePixel   0   0 100
```

Listing 16: Simple Fractals

```
{-
-- Small library that produce P3 ppm images.
-- @author Simon Symeonidis
-}
module Ppm(
PPMImage
, Pixel(Pixel)
, redOf, blueOf, greenOf
, makePixel
, makeImage
, makeWhiteImage, makeBlackImage, makeRedImage, makeBlueImage, makeGreenImage
, outputImage
, setHeader, setWidth, setHeight, setData
, getHeader, getWidth, getHeight, getData
) where

-- | A pixel is a tuple of RGB
data Pixel a = Pixel a a a deriving Show

-- | PPMImage : header, width, height, data
data PPMImage = PPMImage String Int Int [[Pixel Int]]

-- | Header stuff
ppmMagicNumber = "P3"
maxColor = 255
minColor = 0

boundsCheck :: Int -> Int
boundsCheck x =
  case x >= minColor && x <= maxColor of
    True  -> x
    False -> error $ "Pixel values range from 0 to 255, value: " ++ show x

redOf :: Pixel t -> t
redOf (Pixel x _ _) = x

blueOf :: Pixel t -> t
blueOf (Pixel _ _ x) = x

greenOf :: Pixel t -> t
greenOf (Pixel _ x _) = x

whitePixel :: Pixel Int
whitePixel = Pixel maxColor maxColor maxColor

blackPixel :: Pixel Int
```

```haskell
blackPixel = Pixel minColor minColor minColor

redPixel :: Pixel Int
redPixel = Pixel maxColor minColor minColor

bluePixel :: Pixel Int
bluePixel = Pixel minColor maxColor minColor

greenPixel :: Pixel Int
greenPixel = Pixel minColor minColor maxColor

makePixel :: Int -> Int -> Int -> Pixel Int
makePixel r g b = Pixel (boundsCheck r) (boundsCheck g) (boundsCheck b)

--
-- | Image Creation routines
--

-- | Aux function, not to be used
makeRow :: Int -> Pixel t -> [Pixel t]
makeRow 0  _  = []
makeRow it px = px : makeRow (it - 1) px

makeImage :: Pixel Int -> Int -> Int -> PPMImage
makeImage px dimx dimy =
  PPMImage ppmMagicNumber dimx dimy (makeImageData px dimx dimy)

makeImageData :: Pixel Int -> Int -> Int -> [[Pixel Int]]
makeImageData _  _    0    = []
makeImageData px dimx dimy = makeRow dimx px : makeImageData px dimx (dimy - 1)

makeWhiteImage x y = makeImage whitePixel x y
makeBlackImage x y = makeImage blackPixel x y
makeGreenImage x y = makeImage greenPixel x y
makeBlueImage  x y = makeImage bluePixel  x y
makeRedImage   x y = makeImage redPixel   x y

-- | Pixel 0 1 2 -> "0 1 2"
pixelString :: Pixel Int -> String
pixelString pixel =
     (show $ redOf pixel) ++ " "
  ++ (show $ greenOf pixel) ++ " "
  ++ (show $ blueOf pixel)

outputRow :: [Pixel Int] -> String
outputRow [] = ""
outputRow (x:xs) = pixelString x ++ " " ++ outputRow xs

{- Extract the data, and delegate it to some other function
   that knows how to take care of it (in this case this function
   is imageStringBackend -}
imageString :: PPMImage -> String
imageString (PPMImage _ _ _ dat) = imageStringBackend dat

imageStringBackend :: [[Pixel Int]] -> String
imageStringBackend [] = ""
imageStringBackend (row:rows) =
  (outputRow row) ++ "\n" ++ imageStringBackend rows

```

```
outputImage :: PPMImage -> String
outputImage img =
  ppmMagicNumber ++ "\n"
  ++ show (getWidth  img) ++ " "
  ++ show (getHeight img) ++ "\n"
  ++ show (maxColor) ++ "\n"
  ++ (imageString img)

setData :: PPMImage -> [[Pixel Int]] -> PPMImage
setData (PPMImage x y z _) new = PPMImage x y z new

setHeader :: PPMImage -> String -> PPMImage
setHeader (PPMImage _ x y z) new = PPMImage new x y z

setWidth :: PPMImage -> Int -> PPMImage
setWidth (PPMImage x y _ z) new = PPMImage x y new z

setHeight :: PPMImage -> Int -> PPMImage
setHeight (PPMImage x _ y z) new = PPMImage x new y z

getData :: PPMImage -> [[Pixel Int]]
getData (PPMImage _ _ _ d) = d

getHeader :: PPMImage -> String
getHeader (PPMImage h _ _ _) = h

getWidth :: PPMImage -> Int
getWidth (PPMImage _ w _ _) = w

getHeight :: PPMImage -> Int
getHeight (PPMImage _ _ h _) = h
```

Listing 17: PPM Package