Tick 6 — Mandelbrot Set

The aim of this exercise is to draw images containing sections of the Mandelbrot set. We will use an external library for manipulating PNG image files from within ML.

The Teaching Course Information pages for Foundations of Computer Science errata page http://www.cl.cam.ac.uk/teaching/0809/FoundsCS/ERRATA will contain any clarifications. If your issue is still unanswered please send an email to ticks1a-admin@cl.cam.ac.uk. For all other problems you should consult your supervisor in the first instance.

You should hand in your completed submission by 5pm on Wednesday 14th January 2009 in the same manner as the earlier ML ticks. All students will have their ticking session on the following day during the first Java practical of term. You should attend the Java practicals during your allocated time-slot (10am-12pm,12pm-2pm,2pm-4pm or 4pm-6pm).

1 Background Information

The Wikipedia article on the Mandelbrot set is a useful guide and is available here http://en.wikipedia.org/w/index.php?title=Mandelbrot_set&oldid=250859745.[1] This article goes into more depth than is required for this exercise.

The external image library is called Gdimage. This is an example of a library binding—the imaging library libGD is written in C and so the authors of Gdimage have written a (relatively) small amount of stub code which bridges between ML and this C library. This principle could be applied to bind any functionality into an ML program. The Gdimage library is written for Moscow ML and so you need to use this rather than the standard CML interpreter from now on. A list of the libraries for Moscow ML is available here http://www.itu.dk/~sestoft/mosmllib/index.html

Moscow ML is installed on both PWF Windows and Linux. If you wish to install on your own machine, you can download it from http://www.itu.dk/~sestoft/mosml.html. Some Linux distributions and Mac OS X include a version of Moscow ML without external libraries which will therefore not work for this exercise. Your options in this case are to compile from source yourself or to use the PWF Windows or Linux machines. You can remotely connect to a PWF Linux using an ssh client. You should connect to linux.pwf.cl.cam.ac.uk (Access limited to CS undergraduates) or linux.pwf.cam.ac.uk (General access).

2 Exercise

The instructions below will guide you through to the completion of this exercise. You should implement your program in a single source file to which you will add as you progress through the exercise.

1. Prepare a text file called tick6.sml which will contain your source for this exercise. Begin the listing with the following function calls

   ```
   load "Gdimage";
   load "Math";
   ```

   load is a function string -> unit which takes the name of an external library and makes it available to the Moscow ML runtime.

2. Now test that your program successfully loads these libraries. To load your program into the Moscow ML interpreter you should call the function

   ```
   use "c:\\temp\\tick6.sml";
   ```

---

[1]This link is for the current version of the page at the time of writing. The latest version of the page could be completely different, although it is likely to be very similar

Note that you should replace the path above with the entire and complete path to your source file because the ML interpreter will only search for source files in its working directory. If your path contains backslash characters they should be escaped by preceding them with a second backslash.

If your program runs correctly you should see the following output

```
- use "c:\\temp\\tick6.sml";
[opening file "c:\\temp\\tick6.sml"]
> val it = () : unit
> val it = () : unit
[closing file "c:\\temp\\tick6.sml"]
> val it = () : unit
```

3. Consult the API documentation for `Gdimage`. Write a function `blueImage: unit->unit` which writes a completely blue image of 100x100 pixels to the file `blueimage.png` by making calls to the `image` and `toPng` functions. In order to call a function from the `Gdimage` package you should prefix the function name with `Gdimage` and a full-stop. For example, the image function is referred to as `Gdimage.image`.

4. In previous exercises you have seen the function `map` which applies another function to every element of a list, and `maps` (Exercise 5b) which applies a function to every element of a stream. Write a function `mapi` which applies a function to every pixel in a `Gdimage.image`. The function *passed* to `mapi` will be referred to as the colouring function and should have the type

   ```
   int*int -> int*int*int
   ```

   Semantically, this function should take a two-tuple of ints which represents the coordinates of the current pixel and should return a three-tuple of ints which is the RGB value to colour this pixel. Your function `mapi` should thus take a colouring function and colour each pixel of the image according to the output of the colouring function. The final type of `mapi` should be

   ```
   int*int -> int*int*int -> image -> unit.
   ```

   Include a comment in your source file which explains why the `mapi` function returns unit rather than a new image.

5. Add the following function to your program

   ```
   fun gradient (x,y) = (((x div 30) * 30) mod 256,0,((y div 30) * 30) mod 256)
   ```

   Write a function `gradImage:unit->unit` which uses `mapi` and creates an image on disk (`gradient.png`) of dimensions 640x480 pixels with each pixel value set according to the gradient function.[2]

6. Add the following function to your source code. This function checks to see if the point (x,y) lies within the Mandelbrot set. The argument `maxIter` indicates how many attempts should be made before assuming that the point is in the set.

   ```
   fun mandelbrot maxIter (x,y) =
     let
       fun solve (a,b) c =
         if c = maxIter then 1.0
   ```

---

[2](optional): Should you choose to experiment with your own gradient function you will notice the colours in your resultant image do not appear as expected. This is because the implementation of Gdimage makes use of 256-colour images in the underlying image library. This means that if you attempt to draw more than 256 different colours to your image the system will replace existing colours in the image palette with the new ones. Interested students wishing to fix this problem should edit the C source for the Gdimage library (mosml/src/dynlibs/mgd/mgd.c) in the Moscow ML source distribution and replace the call to the C function gdImageCreate with a call to gdImageCreateTrueColor.

```
        else
          if (a*a + b*b <= 4.0) then
            solve (a*a - b*b + x,2.0*a*b + y) (c+1)
          else (real c)/(real maxIter)
   in
     solve (x,y) 0
   end
```

The `mandelbrot` function returns the number of iterations performed expressed as a real number where 0.0 represents zero iterations and 1.0 represents the maximum number of iterations.

7. Add the following function to your source code. This function selects an RGB colour based on the number of iterations returned by the mandelbrot function.

```
fun chooseColour n =
  let
    val r = round ((Math.cos n) * 256.0)
    val g = round ((Math.cos n) * 256.0)
    val b = round ((Math.sin n) * 256.0)
  in
    (r,g,b)
  end
```

The implementations of `sin` and `cos` are provided by the `Math` library which we loaded at the beginning of the program.

8. The `mandelbrot` function does not operate on pixel values. Instead it operates on the number plane. Our image will cover only a portion of the mandelbrot set and so we will need to convert between pixel values and real numbers. Write a function `rescale` with type

   `int*int -> real*real*real -> int*int -> real*real`.

   The following call

   `rescale (w,h) (cx,cy,s) (x,y)`

   should return a tuple (p,q) which is the real number position on the number plane which corresponds to the pixel value (x,y). The tuple (w,h) gives the size of the image in pixels, and the tuple (cx,cy,s) specifies the central point of the image and the size of the window of interest (Figure 1).

   The point p is given as follows (you should derive an equation for q yourself)

$$p = \frac{x}{w} * s - \frac{s}{2} + c_x$$

9. Write a function `compute` which combines your `rescale` function with `mandelbrot` and `chooseColour` in order to compute the Mandelbrot set of a particular region and save the result to a file on disk. Your function `compute` should have the following type

   `real*real*real -> unit`

   where the following invocation

   `compute (cx,cy,s)`

   will return unit and as a side-effect write an image to disk (`mandelbrot.png`) containing the Mandelbrot set for the chosen region (experiment to find a sensible value for `maxIter`).

10. Draw the Mandelbrot set for (cx,cy,s) = (-0.74364990, 0.13188204, 0.00073801)
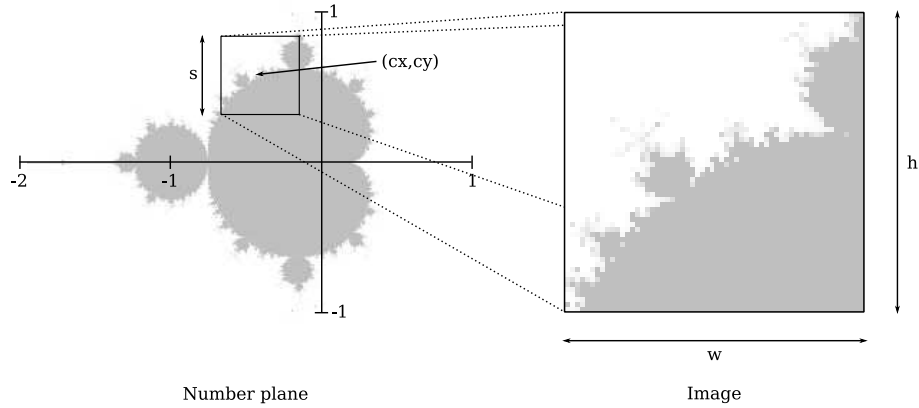
11. Your submission should contain

Figure 1: The function rescale converts a pixel value to a point on the number plane.

- the printed source code as outlined above, including a comment with your name and college in the same manner as the previous exercises, and comments answering the questions as requested in the outline above;

- printouts of the gradient fill test and the selected portion of the Mandelbrot set;

- the amount of time you spent on the exercise.

In order to achieve Tick 6* candidates should further provide a second implementation that can be compiled to a binary using mosmlc. This program should draw arbitrary portions of the Mandelbrot set from Windows Command Prompt (or the Linux shell prompt):

- You should make use of the CommandLine library to recover the options passed to the program by the user.

- The compiler mosmlc will automatically search for external libraries and so the "load" calls at the beginning of your source should be removed.

- It is conventional to call the entry function to your program "main". You can cause your main function to be invoked when the program is executed by ending your source listing with: `val _ = main();`

- The call `C:\temp> mandelbrot -0.74364990 0.13188204 0.00073801 image.png` should draw the required region of the Mandelbrot set to the file image.png.

END OF EXERCISE