University of Copenhagen

Assignment 0: Curves

Jens Fredskov (chw752)

September 8, 2013

1 Introduction

The following report describes the implementation and testing of a Haskell library to work with curves.

2 Implementation

The library has been implemented in the file named `Curve.hs`

2.1 Data structures

The representation of a point has been implemented as:

```
data Point = Point Double Double
```

The structure is straightforward. It could just as well have been:

```
data Point = Point (Double, Double)
```

However the chosen representation allows for more readable code with fewer parentheses. Furthermore we cannot simply define a type alias, as this prevents us from defining `Point` as an instance of `Eq`.

The representation of a curve is simply a type alias to a list of points. The implementation then assumes there is a line segment between every two points,

so that a list $[p_0, p_1, p_2, \ldots, p_{n-1}, p_n]$ has line segments $[p_0p_1, p_1p_2, \ldots, p_{n-1}p_n]$. Had we instead represented the curve in terms of line segments we could have omitted some segments to create a curve with disjoint elements. This however can be achieved in the chosen representation using several curves, and furthermore we also avoid redundancy by not having to specify the same point twice.

2.2 Functions

`connect` could simply connect two curves using the `++`-function. This however means that if one curve ends in the same point as the next begins in we create a redundant line segment of length zero. Two avoid such segments, the function uses a guard to determine if the last point of the first curve and the first point of the second curve are equivalent, and if true removes the first point of the second curve before connecting them. Otherwise the two curves are simply connected.

`reflect` negates the first or second component of every point in the curve (depending of which axis we reflect over) and furthermore adds the point where the axis lies two times, as this corresponds to reflect with respect to an axis running through the point.

`toSvg` uses a helper function `move` which works somewhat similar to translate. The functions translates the curve so that the minimal point of the bounding box lies in $(0, 0)$, as the coordinate system of SVG only handles positive numbers. To generate the line segments the curve is zipped with its tail giving pairs corresponding to the segments.

3 Testing

The curve library has been tested using the functions in `Hilbert.hs`. The file runs the function `hilbert` a given number of times to generate a space-filling curve. The output of running:

```
toFile (hilbertN 4) "./Hilbert.svg"
```

can be seen in figure 1.

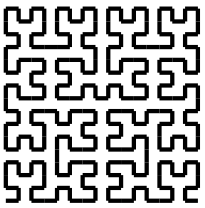

Figure 1: The space-filling curve generated from `Hilbert.hs`

The output corresponds to the expected output. Because the function builds the curve by recursively applying itself on the previous curve, we now that the library must produce a correct Hilbert curve for any number of iterations.

The Hilbert curve tests the creation of points and curves (using the constructor functions). It tests all of the curve manipulation functions (it does not test the case of `reflect` where the axis is vertical, but giving that this corresponds to the horizontal version it we are able to say that it should also be correct). Except for the `toList` (which is trivial) all of the converter functions are used.

The Hilbert curve does not test the guard on `connect`. But running:

```
(curve (point(0,0)) []) `connect` (curve (point(0,0)) [])
```

Which returns `[Point 0.0 0.0]` indicates that this works as intended.

We can thus conclude that the library works correct in our test, seemingly in general.

4 Conclusion

We have now described the implementation of the library, and accounted for design decision with regard to data structures and functions. The testing of the library have been described, and we have concluded that the library has worked correct in all of our tests.