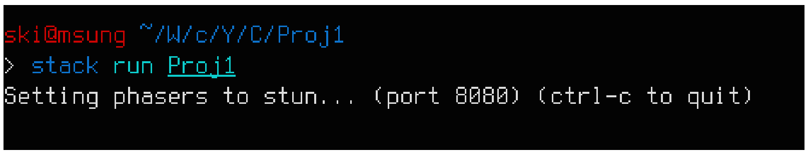
Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44012 Topics In Functional Programming
Project 1 - Image Server

Ted Johnson, TCD 19335618
14th of November, 2022

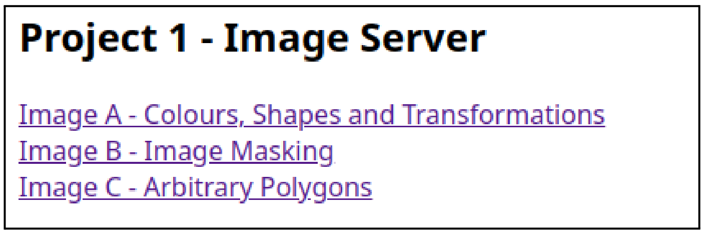
1 Introduction

In this project, I developed an eDSL for drawing shapes and a web server for viewing drawings created with the language. I used the JuicyBits library to produce the PNG graphics specified and Scotty to provide the web server with Blaze for formatting the HTML. I was able to complete all project deliverables to the best of my ability.



```
ski@msung ~/W/c/Y/C/Proj1
> stack run Proj1
Setting phasers to stun... (port 8080) (ctrl-c to quit)
```

Fig 1.1 - The web server running



Project 1 - Image Server

Image A - Colours, Shapes and Transformations
Image B - Image Masking
Image C - Arbitrary Polygons

Fig 1.2 - The root web page on the web server

2 An eDSL for Drawing Shapes

2.1 Design Choices

When deciding on the overall structure of my eDSL, I chose to express drawings as a tree of operations. Every node of the tree would be an action such as applying a transformation or blending in a new colour. A leaf would be the addition of a basic shape to the drawing. Together, a drawing would consist of several basic shapes being augmented by all their parent actions. In this way, basic shapes could 'share' operations. A circle and a square could both be coloured yellow and scaled 40% down by the same chain of parent actions. This felt both intuitive and potentially useful.

Initially, I had considered structuring a drawing as a simple list of shapes, each with their own list of attributes. While this approach would have been easier to implement, it would be much harder to use the eDSL to draw lots of similarly placed or coloured shapes on a drawing - You would have to repeat yourself an awful lot. Sharing operations between shapes like this is also a straightforward optimization technique discussed in section 4.1.

Another design choice was to separate the drawing from the medium - Nowhere in the language should pixels be referenced. Each shape is drawn with infinite detail, it is only the number of pixels contained in the target image which should limit the resolution. Due to this, transformations are always specified relative to the parent node. At the root of the drawing tree, the parent node is considered to be at origin such that the top-left of the target image is equal to (-1,-1) and the bottom-right is equal to (1,1).

To quickly explain the syntax of the language: As a drawing is a tree of operations, each drawing begins as a Haskell list: `[]`. Operations are chained together by embedding them inside sublists: `[ Scale (1.0, 0.2) [ Circle, Square ] ]` - That will draw a circle a square both scaled their y-axises down to 20% (by default, they will both be black and placed at origin). Whitespace is not parsed so it makes much more sense to write more complex drawings like such:

```
[
	Scale (0.5,0.5) [ Colour red [ Square ] ],
	Translate (0.2,0.0) [
		Colour (255,255,100,255) [
			Rotate (0.1) [ Ellipse (0.2,1.0) ],
			Scale (0.25,1.0) [ Square ]
		]
	]
]
```

With this language structure in mind, I did some research into how the JuicyBits library could be used to draw shapes directly onto an image. Previously, we had to test every shape that could potentially influence the colour of a pixel while mapping over every pixel in the image. This was fine for smaller image sizes with a limited number of shapes but it wouldn't be a good fit for my tree traversal design as each basic shape encountered on the tree would have to generate an entirely new image. I solved this issue by editing the image data in-place while traversing the drawing tree with the MutableImage image type. This is mentioned more in section 4.2.

Something I'm not too happy about with the language syntax is the overwhelming presence of square brackets. This is due to how each action node in the tree can have an arbitrary number of children - As such, this is written using the list syntax in Haskell. An improvement to somewhat reduce the square bracket character would be to make it optional when only a single child is being specified. So instead of `[ Rotate 1.5 [ Scale (0.5,0.2) [ Square ] ] ]` it could be possible to just write `[ Rotate 1.5 Scale (0.5,0.2) Square ]` as there's no ambiguity when each parent node has only a single child node.

2.2 Shapes

As mentioned above, every leaf of the drawing tree is a basic shape. These shapes include squares, rectangles, circles, ellipses and polygons. To draw a square and a circle is very simple: `[ Square ]` and `[ Circle ]`. Rectangles require a width and height: `[ Rectangle (0.2,0.5) ]` while the ellipse uses its semi-major and semi-minor axis lengths: `[ Ellipse (0.4,1.0) ]`. Polygons are defined as an ordered list of vertices: `[ Polygon [(0.8,0.8),(-0.6,-0.8),(0.6,-0.8)] ]` . It does not matter the direction of the order nor if the polygon is convex or concave. It's interesting to note that all shapes are internally represented as either an ellipse or polygon for simplicity.

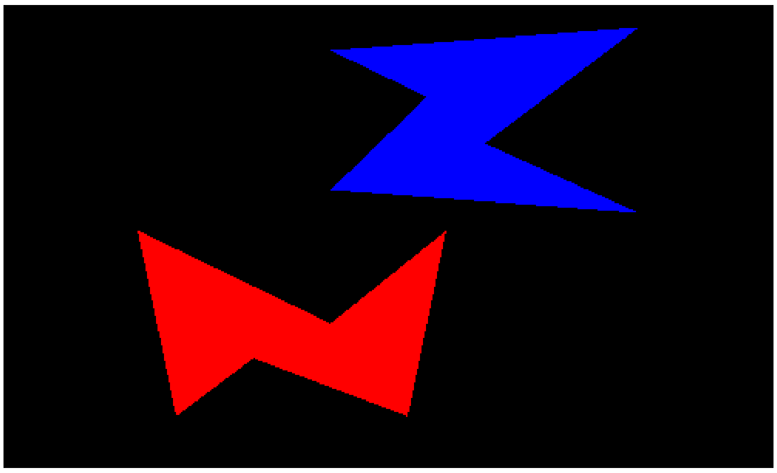


Fig 2.1 - Polygons allow the drawing of arbitrary shapes

2.3 Affine Transformations

To move, rotate and scale the basic shapes around the image, I have provide all of the affine transformations:

```
[
      Translate (x,y) [ children ],
      Rotate angle [ children ],
      Scale (x,y) [ children ],
      Shear (x,y) [ children ]
]
```

New transformations are passed to children by multiplying the corresponding transformation matrix with the current transformation matrix. When it is time to draw a shape, the transformation matrix that has been passed to the leaf node will be used to find the four corners of the bounding box for the shape. Then we use the corresponding shape function to decide which pixels to write to within this bounding box. I discuss in section 4.3 how we use this box to speed up drawing.

2.4 Colours

A colour consists of a red, green, blue and alpha channel. There is an action node for setting the colour of all child nodes: `[ Colour (r,g,b,a) [ children ] ]`. You can also use the colour aliases provided like `[ Colour red [] ]` and `[ Colour blank [] ]`. Non-opaque colours are blended with the parent colour using their alpha channel to create a transparency effect.

2.5 Masking

Drawings can make use of the mask action node to specify two sub-drawings where the first sub-drawing will be masked by the second sub-drawing: `[ Mask [ drawing1 ] [ drawing2 ] ]`. You can influence how much of the first sub-drawing is masked by changing the alpha colour of the shapes in the second sub-drawing.

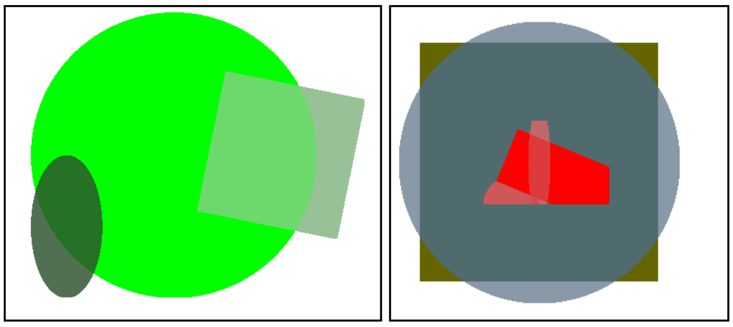


Fig 2.2 - A simple drawings consisting of shapes, colours, transformations and masks

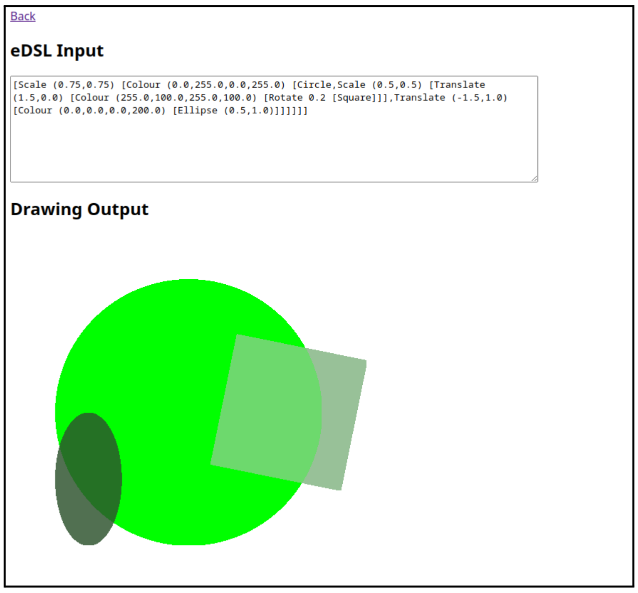
3 The Web Server and UI

3.1 Scotty Web Server

I use the Scotty library to provide a web server to access the images generated by my eDSL. There are two routes specified: A static root route and a variable route for viewing an image alongside its eDSL input. The root web page just links to the 3 different hard-coded images.

3.2 Blaze HTML

The HTML used in this project is very simplistic - Nevertheless, I have used Blaze to properly render and format it.



eDSL Input

```
[Scale (0.75,0.75) [Colour (0.0,255.0,0.0,255.0) [Circle,Scale (0.5,0.5) [Translate
(1.5,0.0) [Colour (255.0,100.0,255.0,100.0) [Rotate 0.2 [Square]]],Translate (-1.5,1.0)
[Colour (0.0,0.0,0.0,200.0) [Ellipse (0.5,1.0)]]]]]]
```

Drawing Output

Fig 3.1 - The layout of the image viewer web page with a hard-coded image

3.3 Image Encoding

One of the earliest challenges I faced in this project was figuring out how to send the image data to the client. The most obvious way was to create a new Scotty route for each image's data alone - Then, like any image used in HTML, the image would be linked to from within the HTML document of the image viewing web page with a src attribute. However, I decided to send all of the image data along with the HTML document at once by encoding the PNG data of the image as a base64 string which, when formatted correctly and placed inside the src attribute, can be interpreted by the browser.

4 Optimisations

4.1 Transformation Hierarchy

The first optimization strategy I implemented was just a result of how the eDSL language is structured. As transformations can be shared between shapes being drawn, it makes sense to only compute these transformation matrices once. Any transformations that are common to several shapes will only be computed once while traversing across the tree. Of course, being a purely functional language, Haskell is already great at performing a computation once and reusing the result.

4.2 MutableImage

A more practical change was to use the MutableImage image type provided by JuicyPixels. By using a stateful monad, it is possible to edit an image in-place while traversing across the drawing tree. This prevents the need to constantly regenerate the entire image every time we render a new shape. It is much faster to just change the corresponding pixels on the same image when rendering a shape. This is especially true when we want to render more than a handful of shapes on our image.

4.3 Bounding Boxes

Finally, as we're editing the image in-place when rendering each shape, it would make sense to only draw to pixels which are actually being changed by the addition of this new shape. We can determine these "relevant" pixels by using the inverse of the affine transformation matrix to find the pixel coordinates of the four most extreme corners of the shape we are about to render. The pixels inside the quad defined by these four pixel coordinates are all of the pixels that could potentially be written to by the drawing of the shape. You can then also discard pixels that lie outside of the image borders. This significantly speeds up the rendering process as smaller shapes only result in a few pixels being written to on the image.

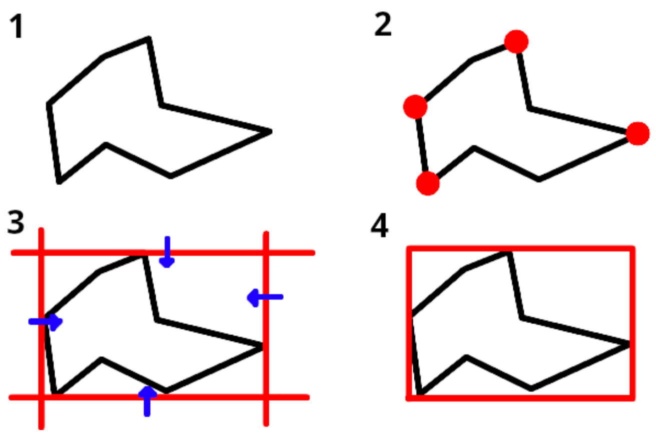


Fig 4.1 - The process of finding the bounding box / relevant pixels for a shape