Exercises on Functional Programming for Domain-Specific Languages

Jeremy Gibbons

Department of Computer Science, University of Oxford
`http://www.cs.ox.ac.uk/jeremy.gibbons/`

Abstract. These exercises have been extracted and slightly updated from my lecture notes on *Functional Programming for Domain-Specific Languages* [1] (p. 1–27 of *Central European Functional Programming Summer School 2013*, Springer Lecture Notes in Computer Science volume 8606, edited by Viktória Zsók, to appear).

4 An extended example: diagrams

We now turn to a larger example of an embedded DSL, inspired by Brent Yorgey's `diagrams` project [2] for two-dimensional diagrams. That project implements a very powerful language which Yorgey does not name, but which we will call *Diagrams*. But it is also rather a large language, so we will not attempt to cover the whole thing; instead, we build a much simpler language in the same spirit.

4.1 Shapes, styles, and pictures

The basics of our diagram DSL can be expressed in three simpler sublanguages, for shapes, styles, and pictures. We represent them via deep embedding. First, there are primitive shapes—as a language, these are not very interesting, because they are non-recursive.

```
data Shape
  = Rectangle Double Double
  | Ellipse Double Double
  | Triangle Double
```

The parameters of a *Rectangle* specify its width and height; those of an *Ellipse* its x- and y-radii. A *Triangle* is equilateral, with its lowest edge parallel to the x-axis; the parameter is the length of the side.

Then there are drawing styles. A *StyleSheet* is a (possibly empty) sequence of stylings, each of which specifies fill colour, stroke colour, or stroke width. (The defaults are for no fill, and very thin black strokes.)

```
type StyleSheet = [Styling]
data Styling
```

$$\begin{aligned} &= \mathit{FillColour}\ \mathit{Col} \\ &\mid \mathit{StrokeColour}\ \mathit{Col} \\ &\mid \mathit{StrokeWidth}\ \mathit{Double} \end{aligned}$$

Here, colours are uninterpreted constants, named according to the W3C SVG Recommendation [3, §4.4].

$$\mathbf{data}\ \mathit{Col} = \mathit{Red} \mid \mathit{Blue} \mid \mathit{Bisque} \mid \mathit{Black} \mid \mathit{Green} \mid \mathit{Yellow} \mid \mathit{Brown}$$

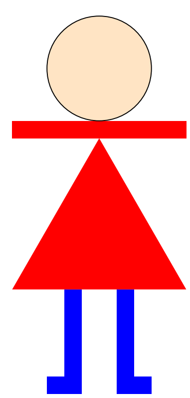
Finally, pictures are arrangements of shapes: individual shapes, with styling; or one picture above another, or one beside another. For simplicity, we specify that horizontal and vertical alignment of pictures is by their centres.

$$\begin{aligned} &\mathbf{data}\ \mathit{Picture} \\ &\quad = \mathit{Place}\ \mathit{StyleSheet}\ \mathit{Shape} \\ &\quad \mid \mathit{Above}\ \mathit{Picture}\ \mathit{Picture} \\ &\quad \mid \mathit{Beside}\ \mathit{Picture}\ \mathit{Picture} \end{aligned}$$

For example, here is a little stick figure of a woman in a red dress and blue stockings.

$$\begin{aligned} &\mathit{figure} :: \mathit{Picture} \\ &\mathit{figure} = \mathit{Place}\ [\mathit{StrokeWidth}\ 0.1, \mathit{FillColour}\ \mathit{bisque}]\ (\mathit{Ellipse}\ 3\ 3)\ `\mathit{Above}` \\ &\qquad\quad \mathit{Place}\ [\mathit{FillColour}\ \mathit{red}, \mathit{StrokeWidth}\ 0]\ (\mathit{Rectangle}\ 10\ 1)\ `\mathit{Above}` \\ &\qquad\quad \mathit{Place}\ [\mathit{FillColour}\ \mathit{red}, \mathit{StrokeWidth}\ 0]\ (\mathit{Triangle}\ 10)\ `\mathit{Above}` \\ &\qquad\quad (\mathit{Place}\ [\mathit{FillColour}\ \mathit{blue}, \mathit{StrokeWidth}\ 0]\ (\mathit{Rectangle}\ 1\ 5)\ `\mathit{Beside}` \\ &\qquad\quad\ \mathit{Place}\ [\mathit{StrokeWidth}\ 0]\ (\mathit{Rectangle}\ 2\ 5)\ `\mathit{Beside}` \\ &\qquad\quad\ \mathit{Place}\ [\mathit{FillColour}\ \mathit{blue}, \mathit{StrokeWidth}\ 0]\ (\mathit{Rectangle}\ 1\ 5))\ `\mathit{Above}` \\ &\qquad\quad (\mathit{Place}\ [\mathit{FillColour}\ \mathit{blue}, \mathit{StrokeWidth}\ 0]\ (\mathit{Rectangle}\ 2\ 1)\ `\mathit{Beside}` \\ &\qquad\quad\ \mathit{Place}\ [\mathit{StrokeWidth}\ 0]\ (\mathit{Rectangle}\ 2\ 1)\ `\mathit{Beside}` \\ &\qquad\quad\ \mathit{Place}\ [\mathit{FillColour}\ \mathit{blue}, \mathit{StrokeWidth}\ 0]\ (\mathit{Rectangle}\ 2\ 1)) \end{aligned}$$

The intention is that it should be drawn like this:



(Note that blank spaces can be obtained by rectangles with no fill and zero stroke width.)

4.2 Transformations

In order to arrange pictures, we will need to be able to translate them. Later on, we will introduce some other transformations too; with that foresight in mind, we introduce a simple language of transformations—the identity transformation, translations, and compositions of these.

```
type Pos = Complex Double

data Transform
  = Identity
  | Translate Pos
  | Compose Transform Transform
```

For simplicity, we borrow the *Complex* type from the Haskell libraries to represent points in the plane; the point with coordinates (x, y) is represented by the complex number $x :+ y$. *Complex* is an instance of the *Num* type class, so we get arithmetic operations on points too. For example, we can apply a *Transform* to a point:

```
transformPos :: Transform → Pos → Pos
transformPos Identity        = id
transformPos (Translate p)   = (p+)
transformPos (Compose t u) = transformPos t ∘ transformPos u
```

Exercises

19. *Transform* is represented above via a deep embedding, with a separate observer function *transformPos*. Reimplement *Transform* via a shallow embedding, with this sole observer.

4.3 Simplified pictures

If we are to export terms in the *Picture* language to a less sophisticated setting—for example, to scalable vector graphics (SVG)—then we eventually have to simplify recursively structured pictures into a flatter form.

Here, we flatten the hierarchy into a non-empty sequence of transformed styled shapes:

```
type Drawing = [(Transform, StyleSheet, Shape)]
```

In order to simplify alignment by centres, we will arrange that each simplified *Drawing* is itself centred: that is, the combined extent of all translated shapes will be centred on the origin. Extents are represented as pairs of points, for the lower left and upper right corners of the orthogonal bounding box.

```
type Extent = (Pos, Pos)
```

The crucial operation on extents is to compute their union:

$$
\begin{array}{l}
\mathit{unionExtent} :: \mathit{Extent} \to \mathit{Extent} \to \mathit{Extent} \\
\mathit{unionExtent}\ (\mathit{llx}_1 :\!+\ \mathit{lly}_1, \mathit{urx}_1 :\!+\ \mathit{ury}_1)\ (\mathit{llx}_2 :\!+\ \mathit{lly}_2, \mathit{urx}_2 :\!+\ \mathit{ury}_2) \\
\quad = (\mathit{min}\ \mathit{llx}_1\ \mathit{llx}_2 :\!+\ \mathit{min}\ \mathit{lly}_1\ \mathit{lly}_2, \mathit{max}\ \mathit{urx}_1\ \mathit{urx}_2 :\!+\ \mathit{max}\ \mathit{ury}_1\ \mathit{ury}_2)
\end{array}
$$

Now, the extent of a drawing is the union of the extents of each of its translated shapes, where the extent of a translated shape is the translation of the two corners of the extent of the untranslated shape:

$$
\begin{array}{l}
\mathit{drawingExtent} :: \mathit{Drawing} \to \mathit{Extent} \\
\mathit{drawingExtent} = \mathit{foldr1}\ \mathit{unionExtent} \circ \mathit{map}\ \mathit{getExtent}\ \mathbf{where} \\
\quad \mathit{getExtent}\ (t, \_, s) = \mathbf{let}\ (\mathit{ll}, \mathit{ur}) = \mathit{shapeExtent}\ s \\
\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ (\mathit{transformPos}\ t\ \mathit{ll}, \mathit{transformPos}\ t\ \mathit{ur})
\end{array}
$$

(You might have thought initially that since all *Drawing*s are kept centred, one point rather than two serves to define the extent. But this does not work: in computing the extent of a whole *Picture*, of course we have to translate its constituent *Drawing*s off-centre.) The extents of individual shapes can be computed using a little geometry:

$$
\begin{array}{ll}
\mathit{shapeExtent} :: \mathit{Shape} \to \mathit{Extent} & \\
\mathit{shapeExtent}\ (\mathit{Ellipse}\ \mathit{xr}\ \mathit{yr}) & = (-(\mathit{xr} :\!+\ \mathit{yr}), \mathit{xr} :\!+\ \mathit{yr}) \\
\mathit{shapeExtent}\ (\mathit{Rectangle}\ w\ h) & = (-({}^{w}\!/_{2} :\!+\ {}^{h}\!/_{2}), {}^{w}\!/_{2} :\!+\ {}^{h}\!/_{2}) \\
\mathit{shapeExtent}\ (\mathit{Triangle}\ s) & = (-({}^{s}\!/_{2} :\!+\ \sqrt{3} \times {}^{s}\!/_{4}), {}^{s}\!/_{2} :\!+\ \sqrt{3} \times {}^{s}\!/_{4})
\end{array}
$$

Now to simplify *Picture*s into *Drawing*s, via a straightforward traversal over the structure of the *Picture*.

$$
\begin{array}{ll}
\mathit{drawPicture} :: \mathit{Picture} \to \mathit{Drawing} & \\
\mathit{drawPicture}\ (\mathit{Place}\ u\ s) & = \mathit{drawShape}\ u\ s \\
\mathit{drawPicture}\ (\mathit{Above}\ p\ q) & = \mathit{drawPicture}\ p\ \text{`}\mathit{aboveD}\text{`}\ \mathit{drawPicture}\ q \\
\mathit{drawPicture}\ (\mathit{Beside}\ p\ q) & = \mathit{drawPicture}\ p\ \text{`}\mathit{besideD}\text{`}\ \mathit{drawPicture}\ q
\end{array}
$$

All the work is in the individual operations. *drawShape* constructs an atomic styled *Drawing*, centred on the origin.

$$
\begin{array}{l}
\mathit{drawShape} :: \mathit{StyleSheet} \to \mathit{Shape} \to \mathit{Drawing} \\
\mathit{drawShape}\ u\ s = [(\mathit{Identity}, u, s)]
\end{array}
$$

aboveD and *besideD* both work by forming the "union" of the two child *Drawing*s, but first translating each child by the appropriate amount—an amount calculated so as to ensure that the resulting *Drawing* is again centred on the origin.

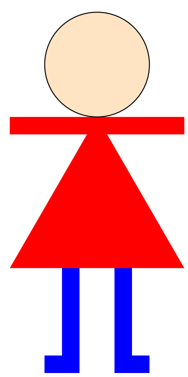
$$
\begin{array}{l}
\mathit{aboveD}, \mathit{besideD} :: \mathit{Drawing} \to \mathit{Drawing} \to \mathit{Drawing} \\
\mathit{pd}\ \text{`}\mathit{aboveD}\text{`}\ \mathit{qd} = \mathit{transformDrawing}\ (\mathit{Translate}\ (0 :\!+\ \mathit{qury}))\ \mathit{pd} +\!\!+ \\
\qquad\qquad\qquad\quad \mathit{transformDrawing}\ (\mathit{Translate}\ (0 :\!+\ \mathit{plly}))\ \mathit{qd}\ \mathbf{where} \\
\quad (\mathit{pllx} :\!+\ \mathit{plly}, \mathit{pur}) = \mathit{drawingExtent}\ \mathit{pd} \\
\quad (\mathit{qll}, \mathit{qurx} :\!+\ \mathit{qury}) = \mathit{drawingExtent}\ \mathit{qd} \\[1ex]
\mathit{pd}\ \text{`}\mathit{besideD}\text{`}\ \mathit{qd} = \mathit{transformDrawing}\ (\mathit{Translate}\ (\mathit{qllx} :\!+\ 0))\ \mathit{pd} +\!\!+ \\
\qquad\qquad\qquad\quad \mathit{transformDrawing}\ (\mathit{Translate}\ (\mathit{purx} :\!+\ 0))\ \mathit{qd}\ \mathbf{where} \\
\quad (\mathit{pll}, \mathit{purx} :\!+\ \mathit{pury}) = \mathit{drawingExtent}\ \mathit{pd} \\
\quad (\mathit{qllx} :\!+\ \mathit{qlly}, \mathit{qur}) = \mathit{drawingExtent}\ \mathit{qd}
\end{array}
$$

This involves transforming the child *Drawing*s; but that is easy, given our representation.

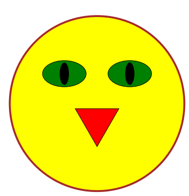
$$\mathit{transformDrawing} :: \mathit{Transform} \rightarrow \mathit{Drawing} \rightarrow \mathit{Drawing}$$
$$\mathit{transformDrawing}\ t = \mathit{map}\ (\lambda(t', u, s) \rightarrow (\mathit{Compose}\ t\ t', u, s))$$

Exercises

20. Add *Square* and *Circle* to the available *Shape*s; for simplicity, you can implement these using *rect* and *ellipseXY*.
21. Add *Blank* to the available shapes; implement this as a rectangle with no fill and stroke width zero.
22. Centring and alignment, as described above, are only approximations, because we do not take stroke width into account. How would you do so?
23. Add *InFrontOf* :: *Picture* → *Picture* → *Picture* as an operator to the *Picture* language, for placing one *Picture* in front of (that is, on top of) another. Using this, you can draw a slightly less childish-looking stick figure, with the "arms" overlaid on the "body":



24. Add *FlipV* :: *Picture* → *Picture* as an operator to the *Picture* language, for flipping a *Picture* vertically (that is, from top to bottom, about a horizontal axis). Then you can draw this chicken:



You will need to add a corresponding operator *ReflectY* to the *Transform* language; you might note that the *conjugate* function on complex numbers takes $x :+ y$ to $x :+ (-y)$. Be careful in computing the extent of a flipped picture!

25. *Picture* is represented above via a deep embedding, with a separate observer function *drawPicture*. Reimplement *Picture* via a shallow embedding, with this sole observer.

4.4 Generating SVG

The final step is to translate our simplified *Drawing* into SVG (which is a dialect of XML). What we need are the following:

– a datatype for representing XML:

$$\begin{array}{l}\textbf{data}\ \mathit{XML} = \mathit{Element}\ \mathit{String}\ [\mathit{Attr}]\ [\mathit{XML}] \\ \textbf{type}\ \mathit{Attr} = (\mathit{String}, \mathit{String})\end{array}$$

– a rendering of *StyleSheet*s as attribute lists:

$$\mathit{applyStyleSheet} :: \mathit{StyleSheet} \rightarrow [\mathit{Attr}]$$

– a rendering of individual transformed styled shapes as XML:

$$\mathit{diagramShape} :: (\mathit{Transform}, \mathit{StyleSheet}, \mathit{Shape}) \rightarrow \mathit{XML}$$

– functions to compute the top-level attributes for a diagram:

$$\mathit{drawingAttrs}, \mathit{groupAttrs} :: \mathit{Drawing} \rightarrow [\mathit{Attr}]$$

– a wrapper function that writes out an XML element to a specified file:

$$\mathit{writeSVG} :: \mathit{FilePath} \rightarrow \mathit{XML} \rightarrow \mathit{IO}\ ()$$

Then a *Drawing* can be assembled into XML:

$$\begin{array}{l}\mathit{assemble} :: \mathit{Drawing} \rightarrow \mathit{XML} \\ \mathit{assemble}\ d = \mathit{Element}\ \texttt{"svg"}\ (\mathit{drawingAttrs}\ d) \\ \quad [\mathit{Element}\ \texttt{"g"}\ (\mathit{groupAttrs}\ d)\ (\mathit{map}\ \mathit{diagramShape}\ d)]\end{array}$$

and subsequently written to an SVG file. And that is it! (You can look at the source file `Shapes.lhs` for the gory details.)

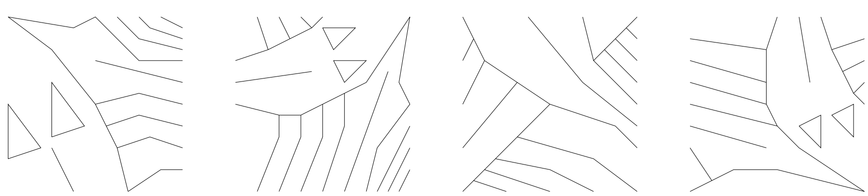
Exercises

26. In Exercise 19, we reimplemented *Transform* as a shallow embedding, with the sole observer being to transform a point. This does not allow us to apply the same transformations directly to *Drawing*s, as required by the function *transformDrawing* above—instead, we have to take a *Drawing* apart to manipulate the *Transform* inside. Extend the shallow embedding of *Transform* so that it has two observers, for transforming both points and drawings.
27. A better solution to Exercise 26 would be to represent *Transform* via a shallow embedding with a single parametrised observer, which can be instantiated at least to the two uses we require. What are the requirements on such instantiations?

28. Simplifying a *Picture* into a *Drawing* is a bit inefficient, because we have to continually recompute extents. A more efficient approach would be to extend the *Drawing* type so that it caches the extent, as well as storing the list of shapes. Try this.
29. It can be a bit painful to specify a complicated *Picture* with lots of *Shape*s all drawn in a common style—for example, all blue, with a thick black stroke—because those style settings have to be repeated for every single *Shape*. Extend the *Picture* language so that *Picture*s too may have *StyleSheet*s; styles should be inherited by children, unless they are overridden. SVG `<g>` elements group together a collection of children, and styling of the group is similarly inherited by its children.
30. Add an operator *Tile* to the *Shape* language, for square tiles with markings on. It should take a *Double* parameter for the length of the side, and a list of lists of points for the markings; each list of points has length at least two, and denotes a path of straight-line segments between those points. For example, here is one such pattern of markings:

$$
\begin{aligned}
&\mathit{markingsP} :: [[\mathit{Pos}]] \\
&\mathit{markingsP} = [[(4 :+ 4), (6 :+ 0)], \\
&\qquad [(0 :+ 3), (3 :+ 4), (0 :+ 8), (0 :+ 3)], \\
&\qquad [(4 :+ 5), (7 :+ 6), (4 :+ 10), (4 :+ 5)], \\
&\qquad [(11 :+ 0), (10 :+ 4), (8 :+ 8), (4 :+ 13), (0 :+ 16)], \\
&\qquad [(11 :+ 0), (14 :+ 2), (16 :+ 2)], \\
&\qquad [(10 :+ 4), (13 :+ 5), (16 :+ 4)], \\
&\qquad [(9 :+ 6), (12 :+ 7), (16 :+ 6)], \\
&\qquad [(8 :+ 8), (12 :+ 9), (16 :+ 8)], \\
&\qquad [(8 :+ 12), (16 :+ 10)], \\
&\qquad [(0 :+ 16), (6 :+ 15), (8 :+ 16), (12 :+ 12), (16 :+ 12)], \\
&\qquad [(10 :+ 16), (12 :+ 14), (16 :+ 13)], \\
&\qquad [(12 :+ 16), (13 :+ 15), (16 :+ 14)], \\
&\qquad [(14 :+ 16), (16 :+ 15)] \\
&\qquad]
\end{aligned}
$$

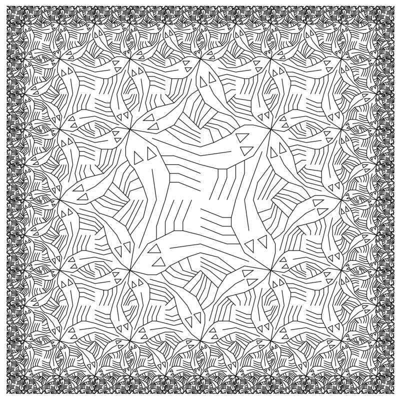
In `Shapes.lhs`, you will find this definition plus three others like it. They yield tile markings looking like this:

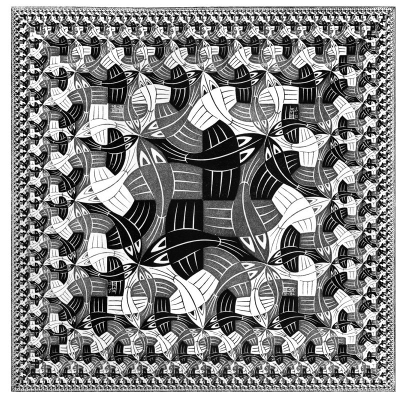


You can render such tiles via the function

$$\mathit{polyline} :: [\mathit{Pos}] \to \mathit{XML}$$

provided for you. Also add operators to the *Picture* and *Transform* languages to support scaling by a constant factor and rotation by a quarter-turn anticlockwise, both centred on the origin. Then suitable placements, rotations, and scalings of the four marked tiles will produce a rough version of Escher's "Square Limit" print, as shown in the left-hand image below:





This construction was explored by Peter Henderson in a famous early paper on functional geometry [4, 5]; I have taken the data for the markings from a note by Frank Buß [6]. The image on the right is the the real "Square Limit" [7].

31. Morally, "Square Limit" is a fractal image: the recursive decomposition can be taken ad infinitum. Because Haskell uses lazy evaluation, that is not an insurmountable obstacle. The datatype *Picture* includes also infinite terms; and because *Diagrams* is an embedded DSL, you can use a recursive Haskell definition to define an infinite *Picture*. You cannot render it directly to SVG, though; that would at best yield an infinite SVG file. But still, you can prune the infinite picture to a finite depth, and then render the result. Construct the infinite *Picture*. (You will probably need to refactor some code. Note that you cannot compute the extent of an infinite *Picture* either—how can you get around that problem?)

References

1. Gibbons, J.: Functional programming for domain-specific languages. In Zsók, V., ed.: Central European Functional Programming Summer School, July 2013. Volume 8606 of Lecture Notes in Computer Science., Springer (2014) 1–27 To appear.
2. Yorgey, B.: Diagrams 0.6. `http://projects.haskell.org/diagrams/` (2012)
3. W3C: Scalable vector graphics (SVG) 1.1: Recognized color keyword names. `http://www.w3.org/TR/SVG11/types.html#ColorKeywords` (2011)
4. Henderson, P.: Functional geometry. In: Lisp and Functional Programming. (1982) 179–187 `http://users.ecs.soton.ac.uk/ph/funcgeo.pdf`.
5. Henderson, P.: Functional geometry. Higher Order and Symbolic Computing **15**(4) (2002) 349–365 Revision of [4].
6. Buß, F.: Functional geometry. `http://www.frank-buss.de/lisp/functional.html` (2005)
7. Escher, M.C.: Square limit. `http://www.wikipaintings.org/en/m-c-escher/square-limit` (1964)