Diagrams: A Functional EDSL for Vector Graphics

Ryan Yates
Department of Computer Science
University of Rochester
Rochester, New York, USA
ryates@cs.rochester.edu

Brent A. Yorgey
Dept. of Mathematics and Computer Science
Hendrix College
Conway, Arkansas, USA
byorgey@gmail.com

Abstract

diagrams is a domain-specific language for creating vector graphics. We will give a short diagrams tutorial/demo, particularly highlighting the power of a functional, embedded domain-specific language.

Categories and Subject Descriptors I.3.6 [*Computer Graphics*]: Methodology and Techniques—Languages; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Languages

Keywords diagrams, Haskell, EDSL, vector

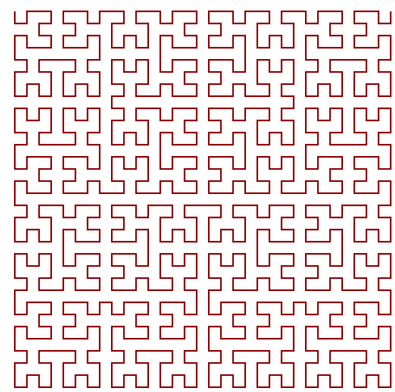
1. Introduction

diagrams (`http://projects.haskell.org/diagrams`) is a declarative domain-specific language for creating vector graphics. embedded in the Haskell programming language (Marlow 2010). Under continuous development for the past 4+ years, it serves as a powerful platform for creating illustrations, visualizations, and artwork, as well as a testbed for new ideas in functional EDSLs and in functional approaches to graphics. Designed with “power users” in mind, it includes support for multiple vector spaces, pluggable rendering backends, sophisticated algorithms for working with paths, and relative positioning of the constituent parts of a diagram. It makes extensive use of Haskell’s type system to capture geometric invariants, and uses a pure functional paradigm both in its internal design (for example, using first-class functions to represent information about boundaries) as well as in the design of its API, which emphasizes composition rather than mutation.

We will begin by explaining just enough of the basics to get started, and then use the remainder of the time to show off some more sophisticated examples. In what follows, we include a few representative examples, with commentary explaining what features of the framework are illustrated by each example, and the particular ways in which the examples highlight the power of a functional EDSL (Hudak 1996).

This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

FARM’15, September 5, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3806-6/15/09
http://dx.doi.org/10.1145/2808083.2808085



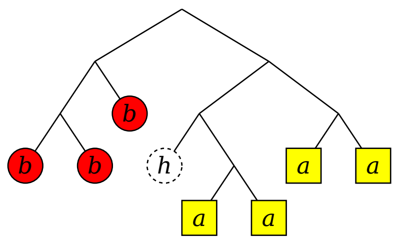
```
hilbert 0 = mempty
hilbert n = hilbert' (n-1) # reflectY <> vrule 1
         <> hilbert  (n-1) <> hrule 1
         <> hilbert  (n-1) <> vrule (-1)
         <> hilbert' (n-1) # reflectX
  where
    hilbert' m = hilbert m # rotateBy (1/4)

dia = hilbert 5 # strokeT
    # lc darkred # lw medium # frame 1
```

Figure 1. Order-5 Hilbert curve, with code

2. Examples

Figure 1 shows an order-5 fractal Hilbert curve (Hilbert 1891), along with the complete code used to generate it. Of course, recursive functions such as `hilbert` are the bread and butter of functional programming. This example also shows off the compositional nature of the framework, in this case building up a complex *path* by concatenating shorter paths using the `<>` operator. In fact, `<>` denotes not just concatenation of paths, but more generally the associative combining operation for any *monoid*—of which diagrams has quite a few, including paths, colors, transformations, styles, and diagrams themselves (Yorgey 2012).

Figure 2 shows a leaf-labelled binary tree along with the complete code used to generate it (Piponi and Yorgey 2015). The first few lines define `t`, an abstract representation of the tree to be drawn, and the rest of the lines specify how to render it. This example illustrates the ability of an *embedded* DSL to leverage the abstraction facilities of its host language. Here we define a new data type, `LeafType`, and use it to enumerate the possibilities for leaves in the tree to be drawn. We define functions to abstract out common pat-



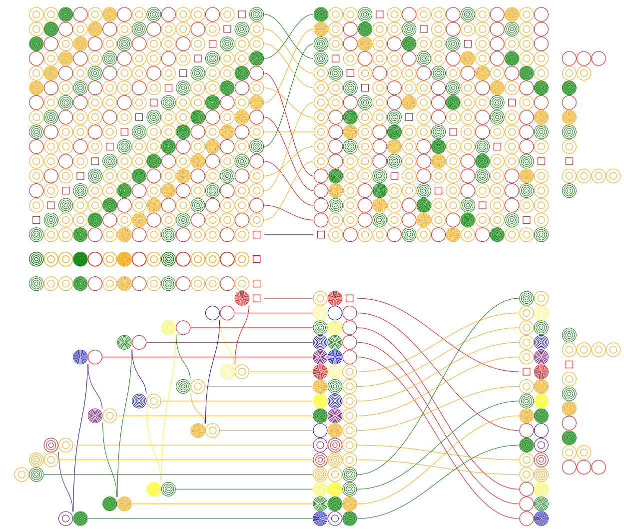

```
import Diagrams.TwoD.Layout.Tree
import Data.Tree
import Data.Char (toLower)

data LeafType = A | B | H  deriving Show

t = nd [ nd [ nd $ map lf [B, B], lf B ]
       , nd [ nd [ lf H, nd $ map lf [A, A] ]
            , nd $ map lf [A, A]
            ]
       ]
  where nd     = Node Nothing
        lf x   = Node (Just x) []

drawType x = mconcat
  [ text (map toLower (show x)) # italic # centerX
  , drawNode x ]

drawNode A = square 2 # fc yellow
drawNode B = circle 1 # fc red
drawNode H = circle 1 # fc white
                      # dashingG [0.2,0.2] 0

renderT :: Tree (Maybe LeafType) -> Diagram B
renderT
  = renderTree (maybe mempty drawType) (~~)
  . symmLayout' (with & slHSep .~ 4 & slVSep .~ 3)

dia = renderT t # frame 0.5
```

Figure 2. Labelled binary tree, with code

terns (`nd`, `lf`) and to specify custom behavior (`drawType`). We also make use of higher-order functions: `map` is higher-order, of course, but more interestingly, so is `renderTree`, which takes function arguments specifying how to draw nodes and edges of a tree. Finally, this example shows off the fact that diagrams comes with "batteries included", such as the tree layout algorithm used here.

Finally, Figure 3 shows a portrait of the Burrows–Wheeler transform (BWT) (Burrows and Wheeler 1994) that was included in the Bridges mathematical art exhibition at the 2014 Joint Mathematics Meetings. This is a portrait in the sense that it captures an aspect of an algorithm concretely. In the middle on the left side an input value starts the diagram. This value is manipulated according to the steps of BWT clockwise with the top half encoding and the bottom half decoding back to the original inpnut.

Having the full expressiveness of Haskell helped to shape this work as it was created. Processes like extracting common code and generalizing functions allowed rapid exploration of visual patterns and the development of a visual language for the work. For instance, the `alphabet` function produces a diagram of nested circles for a given number. Originally it had the colors "baked in", but later, when connecting parts of the diagram, Haskell's ease of



```
inputToBWT =
  [ block rs # reflectX      -- Rotations of s
  , sorting 7 head rs rs'
  , block rs'                -- Sorted rotations
                             -- of s
  ]
  # map centerXY
  # hcat' (with & sep .~ 0.1)
```

Figure 3. A portrait of the Burrows–Wheeler transform. The small code fragment generates the top portion of the image.

refactoring allowed the extraction of a function with the colors with little effort.

The flexibility of diagrams also allowed the exploration of various compositions with little change to the code. Instead of blocks proceeding clockwise, we could have a single linear progression, or a radial layout that fanned out like a circle. Most of these variations could be explored with small changes to the code responsible for composition. Indeed, various layouts were revisited later in the process even with significant changes in the code by keeping the other layouts around and fixing errors caught by type checking. In the end the algorithm's own transition from row to column gives an opportunity for the arrangement around a square and a reflective symmetry of sorts across the horizontal midline.

References

Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

Simon Marlow. Haskell 2010 Language Report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

Dan Piponi and Brent A Yorgey. Polynomial functors constrained by regular expressions. In *Proceedings of the Mathematics of Program Construction*, 2015.

Brent A Yorgey. Monoids: theme and variations (functional pearl). In *ACM SIGPLAN Notices*, volume 47, pages 105–116. ACM, 2012.