Declarative, Programmatic Vector Graphics in Haskell

Brent Yorgey

Libre Graphics Meeting
Leipzig
3 April, 2013
Embedded in Haskell.
circle 1 # fc green ||| square 2 # fc blue
circle 1 # fc green || square 2 # fc blue

Look ma, no coordinates!
fib 0 = leaf 0; fib 1 = leaf 1
fib n = BNode n (fib (n-1)) (fib (n-2))

tree
  = renderTree' 
    (\i -> circle 0.3 # lw 0 # fc (colors !! i)) 
    (\(i,p) (_,q) -> p ~~ q # lc (colors !! i)) 
    . fromJust . symmLayoutBin $ fib 8
Haskell and EDSLs

Haskell makes a great host language for DSLs:

- strong static type system
- first-class functions
- powerful abstraction mechanisms
- culture that encourages elegant, mathematically-based design: theory meets practice
Haskell and EDSLs

Haskell makes a great host language for DSLs:

- strong static type system
- first-class functions
- powerful abstraction mechanisms
- culture that encourages elegant, mathematically-based design: theory meets practice

Full disclosure:

- Error messages suck
Haskell has a **strong static type system**.
Haskell has a **strong static type system**.

Impossible to make silly mistakes like applying a vector to a color, or adding two points.
Haskell has **first-class functions**.
Haskell has **first-class functions**.
Abstraction

Haskell has powerful abstraction mechanisms.
Haskell has **powerful abstraction mechanisms**.

\[
\text{square} :: \text{Double} \rightarrow \text{Diagram}
\]
Haskell has **powerful abstraction mechanisms**.

```
square :: (TrailLike t, Transformable t, V t ~ R2)
  => Double -> t
```
Design

Haskell encourages **elegant, mathematically-based design**.
Haskell encourages **elegant, mathematically-based design**.
Monoids: Theme and Variations (Functional Pearl)

Brent A. Yorgey
University of Pennsylvania
byorgey@cis.upenn.edu

Abstract
The monoid is a humble algebraic structure, at first glance even downright boring. However, there’s much more to monoids than meets the eye. Using examples taken from the diagrams vector graphics framework as a case study, I demonstrate the power and beauty of monoids for library design. The paper begins with an extremely simple model of diagrams and proceeds through a series of incremental variations, all related somehow to the central theme of monoids. Along the way, I illustrate the power of compositional semantics; why you should also pay attention to the monoid’s even humbler cousin, the semigroup; monoid homomorphisms; and monoid actions.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming. D.2.2 [Design Tools and Techniques]

General Terms Languages, Design

Keywords monoid, homomorphism, monoid action, EDSL

Prelude
diagrams is a framework and embedded domain-specific language for creating vector graphics in Haskell. All the illustrations in this paper were produced using diagrams, and all the examples inspired by it. However, this paper is not really about diagrams at all! It is really about monoids, and the powerful role they—and, more generally, any mathematical abstraction—can play in library design. Although diagrams is used as a specific case study, the central ideas are applicable in many contexts.

Theme
What is a diagram? Although there are many possible answers to this question (examples include those of Elliott [2003] and Matlage and Gill [2011]), the particular semantics chosen by diagrams is an ordered collection of primitives. To record this idea as Haskell code, one might write:

```
type Diagram = [Prim]
```

But what is a primitive? For the purposes of this paper, it doesn’t matter. A primitive is a thing that Can Be Drawn—like a circle, arc, polygon, Bézier curve, and so on—and inherently possesses any attributes we might care about, such as color, size, and location.

The primitives are ordered because we need to know which should appear “on top.” Concretely, the list represents the order in which the primitives should be drawn, beginning with the “bottommost” and ending with the “topmost” (see Figure 1). Lists support concatenation, and “concatenating” two Diagrams also makes good sense: concatenation of lists of primitives corresponds to superposition of diagrams—that is, placing one diagram on top of another. The empty list is an identity element for concatenation ([ ] ++ [ ] = [ ], and this makes sense in the context of diagrams as well: the empty list of primitives represents the empty diagram, which is an identity element for superposition. List concatenation is associative; diagram A on top of (diagram B on top of C) is the same as (A on top of B) on top of C. In short, (++) and ++ constitute a monoidal structure on lists, and hence on diagrams as well.

This is an extremely simple representation of diagrams, but it already illustrates why monoids are so fundamentally important: composition is at the heart of diagrams—and, indeed, of many libraries. Putting one diagram on top of another may not seem very expressive, but it is the fundamental operation out of which all other modes of composition can be built.

However, this really is an extremely simple representation of diagrams—much too simple! The rest of this paper develops a series of increasingly sophisticated variant representations for Diagram, each using a key idea somehow centered on the theme of monoids. But first, we must take a step backwards and develop this underlying theme itself.

Interlude
The following discussion of monoids—and the rest of the paper in general—relies on two simplifying assumptions:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

\textcopyright{} 2012 ACM 978-1-4503-1574-6/12/09 ... $10.00
What’s next?
What’s next?
What’s next?
What’s next?

- Google Summer of Code project to allow editing diagrams.
What’s next?

- Google Summer of Code project to allow **editing** diagrams.
- Animations and interactivity.
What’s next?

- Google Summer of Code project to allow editing diagrams.
- Animations and interactivity.
- Bidirectional GUI/code editor.
What’s next?

- Google Summer of Code project to allow *editing* diagrams.
- Animations and interactivity.
- Bidirectional GUI/code editor.
- Open to suggestions!
http://projects.haskell.org/diagrams
Extra slides
Backends

and:

- OpenGL
- HTML5 canvas
- PGF/TikZ
- PDF
- native Haskell raster library
shapes = hcat' (with & sep .~ 3) 
    [ square 2 # fc green # named "s"
      , circle 1 # fc blue # named "c"
    ]
dia = shapes 
    # connectOutside' (with & gap .~ 0.2) "s" "c"
```haskell
dia = hcat' (with & sep ~ 1)
  [ square 1
    , mconcat
      [ square 1
        , square 1 # reversePath # rotateBy (1/7)
      ]
    # stroke # fc red
    , square 1 # map (place dot) # mconcat
  ]

where
  dot = circle 0.2 # fc blue # lw 0
```