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.[1] 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.

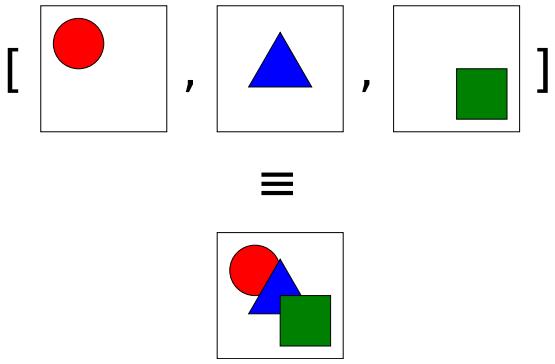

Figure 1. Superimposing a list of primitives

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 ($[\,] +\!\!+ xs = xs +\!\!+ [\,] = xs$), 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 *monoid* 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:

[1] `http://projects.haskell.org/diagrams/`

Haskell '12, September 13, 2012, Copenhagen, Denmark.

- all values are finite and total;
- the floating-point type Double is a well-behaved representation of the real numbers $\mathbb{R}$.

The first assumption is reasonable, since we will have no need for infinite data structures, nontermination, or partial functions. The second is downright laughable, but makes up for in convenience what it lacks in accuracy.

Monoids

A *monoid* is a set S along with a binary operation $\diamond :: S \to S \to S$ and a distinguished element $\varepsilon :: S$, subject to the laws

$$\varepsilon \diamond x = x \diamond \varepsilon = x \tag{M1}$$

$$x \diamond (y \diamond z) = (x \diamond y) \diamond z. \tag{M2}$$

where x, y, and z are arbitrary elements of S. That is, ε is an *identity* for $\diamond$ (M1), which is required to be *associative* (M2).

Monoids are represented in Haskell by the Monoid type class defined in the `Data.Monoid` module, which is part of the standard `base` package.

```
class Monoid a where
  ε      :: a
  (◇)    :: a → a → a
  mconcat :: [a] → a
  mconcat = foldr (◇) ε
```

The actual Monoid methods are named *mempty* and *mappend*, but I will use ε and $(\diamond)$ in the interest of brevity.

mconcat "reduces" a list using $(\diamond)$, that is,

$$mconcat\ [a,b,c,d] = a \diamond (b \diamond (c \diamond d)).$$

It is included in the Monoid class in case some instances can override the default implementation with a more efficient one.

At first, monoids may seem like too simple of an abstraction to be of much use, but associativity is powerful: applications of *mconcat* can be easily parallelized [Cole 1995], recomputed incrementally [Piponi 2009], or cached [Hinze and Paterson 2006]. Moreover, monoids are ubiquitous—here are just a few examples:

- As mentioned previously, lists form a monoid with concatenation as the binary operation and the empty list as the identity.
- The natural numbers $\mathbb{N}$ form a monoid under both addition (with 0 as identity) and multiplication (with 1 as identity). The integers $\mathbb{Z}$, rationals $\mathbb{Q}$, real numbers $\mathbb{R}$, and complex numbers $\mathbb{C}$ all do as well. `Data.Monoid` provides the Sum and Product **newtype** wrappers to represent these instances.
- $\mathbb{N}$ also forms a monoid under *max* with 0 as the identity. However, it does not form a monoid under *min*; no matter what $n \in \mathbb{N}$ we pick, we always have $min(n, n+1) = n \neq n+1$, so n cannot be the identity element. More intuitively, an identity for *min* would have to be "the largest natural number", which of course does not exist. Likewise, none of $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ form monoids under *min* or *max* (and *min* and *max* are not even well-defined on $\mathbb{C}$).
- The set of booleans forms a monoid under conjunction (with identity True), disjunction (with identity False) and exclusive disjunction (again, with identity False). `Data.Monoid` provides the All and Any **newtype** wrappers for the first two instances.
- Sets, as defined in the standard Data.Set module, form a monoid under set union, with the empty set as the identity.
- Given Monoid instances for m and n, their product (m,n) is also a monoid, with the operations defined elementwise:

```
instance (Monoid m, Monoid n)
      ⇒ Monoid (m,n) where
  ε             = (ε, ε)
  (m1,n1) ◇ (m2,n2) = (m1 ◇ m2, n1 ◇ n2)
```

- A function type with a monoidal result type is also a monoid, with the results of functions combined pointwise:

```
instance Monoid m ⇒ Monoid (a → m) where
  ε       = const ε
  f1 ◇ f2 = λa → f1 a ◇ f2 a
```

  In fact, if you squint and think of the function type $a \to m$ as an "a-indexed product" of m values, you can see this as a generalization of the instance for binary products. Both this and the binary product instance will play important roles later.
- Endofunctions, that is, functions $a \to a$ from some type to itself, form a monoid under function composition, with the identity function as the identity element. This instance is provided by the Endo **newtype** wrapper.
- The *dual* of any monoid is also a monoid:

```
newtype Dual a = Dual a

instance Monoid a ⇒ Monoid (Dual a) where
  ε                   = Dual ε
  (Dual m1) ◇ (Dual m2) = Dual (m2 ◇ m1)
```

  In words, given a monoid on a, Dual a is the monoid which uses the same binary operation as a, but with the order of arguments switched.

Finally, a monoid is *commutative* if the additional law

$$x \diamond y = y \diamond x$$

holds for all x and y. The reader can verify how commutativity applies to the foregoing examples: Sum, Product, Any, and All are commutative (as are the *max* and *min* operations); lists and endofunctions are not; applications of $(,)$, $((\to)\ e)$, and Dual are commutative if and only if their arguments are.

Monoid homomorphisms

A *monoid homomorphism* is a function from one monoidal type to another which preserves monoid structure; that is, a function f satisfying the laws

$$f\ \varepsilon = \varepsilon \tag{H1}$$

$$f\ (x \diamond y) = f\ x \diamond f\ y \tag{H2}$$

For example, $length\ [\,] = 0$ and $length\ (xs +\!\!+ ys) = length\ xs + length\ ys$, making *length* a monoid homomorphism from the monoid of lists to the monoid of natural numbers under addition.

Free monoids

Lists come up often when discussing monoids, and this is no accident: lists are the "most fundamental" Monoid instance, in the precise sense that the list type $[a]$ represents the *free monoid* over a. Intuitively, this means that $[a]$ is the result of turning a into a monoid while "retaining as much information as possible". More formally, this means that any function $f :: a \to m$, where m is a monoid, extends uniquely to a monoid homomorphism from $[a]$ to m—namely, $mconcat \circ map\ f$. It will be useful later to give this construction a name:

```
hom :: Monoid m ⇒ (a → m) → ([a] → m)
hom f = mconcat ∘ map f
```

See the Appendix for a proof that *hom f* really is a monoid homomorphism.

Semigroups

A *semigroup* is like a monoid *without* the requirement of an identity element: it consists simply of a set with an associative binary operation.

Semigroups can be represented in Haskell by the Semigroup type class, defined in the `semigroups` package[2]:

```
class Semigroup a where
  (◇) :: a → a → a
```

(The Semigroup class also declares two other methods with default implementations in terms of (◇); however, they are not used in this paper.) The behavior of Semigroup and Monoid instances for the same type will always coincide in this paper, so using the same name for their operations introduces no ambiguity. I will also pretend that Monoid has Semigroup as a superclass, although in actuality it does not (yet).

One important family of semigroups which are *not* monoids are unbounded, linearly ordered types (such as $\mathbb{Z}$ and $\mathbb{R}$) under the operations of *min* and *max*. `Data.Semigroup` defines Min as

```
newtype Min a = Min {getMin :: a}
instance Ord a ⇒ Semigroup (Min a) where
  Min a ◇ Min b = Min (min a b)
```

and Max is defined similarly.

Of course, any monoid is automatically a semigroup (by forgetting about its identity element). In the other direction, to turn a semigroup into a monoid, simply add a new distinguished element to serve as the identity, and extend the definition of the binary operation appropriately. This creates an identity element by definition, and it is not hard to see that it preserves associativity.

In some cases, this new distinguished identity element has a clear intuitive interpretation. For example, a distinguished identity element added to the semigroup $(\mathbb{N}, min)$ can be thought of as "positive infinity": $min(+\infty, n) = min(n, +\infty) = n$ for all natural numbers n.

Adding a new distinguished element to a type is typically accomplished by wrapping it in Maybe. One might therefore expect to turn an instance of Semigroup into an instance of Monoid by wrapping it in Maybe. Sadly, `Data.Monoid` does not define semigroups, and has a Monoid instance for Maybe which requires a Monoid constraint on its argument type:

```
instance Monoid a ⇒ Monoid (Maybe a) where
  ε = Nothing
  Nothing ◇ b       = b
  a       ◇ Nothing = a
  (Just a) ◇ (Just b) = Just (a ◇ b)
```

This is somewhat odd: in essence, it ignores the identity element of a and replaces it with a different one. As a workaround, the `semigroups` package defines an Option type, isomorphic to Maybe, with a more sensible Monoid instance:

```
newtype Option a = Option {getOption :: Maybe a}
instance Semigroup a ⇒ Monoid (Option a) where
  ...
```

The implementation is essentially the same as that for Maybe, but in the case where both arguments are Just, their contents are combined according to their Semigroup structure.

Variation I: Dualizing diagrams

Recall that since Diagram is (so far) just a list, it has a Monoid instance: if d_1 and d_2 are diagrams, then $d_1 \diamond d_2$ is the diagram containing the primitives from d_1 followed by those of d_2. This means that d_1 will be drawn first, and hence will appear *beneath* d_2. Intuitively, this seems odd; one might expect the diagram which comes first to end up on top.

Let's define a different Monoid instance for Diagram, so that $d_1 \diamond d_2$ will result in d_1 being on top. First, we must wrap [Prim] in a **newtype**. We also define a few helper functions for dealing with the **newtype** constructor:

```
newtype Diagram = Diagram [Prim]

unD :: Diagram      → [Prim]
unD  (Diagram ps) = ps

prim :: Prim → Diagram
prim  p      = Diagram [p]

mkD :: [Prim] → Diagram
mkD = Diagram
```

And now we must tediously declare a custom Monoid instance:

```
instance Monoid Diagram where
  ε                                = Diagram ε
  (Diagram ps1) ◇ (Diagram ps2) = Diagram (ps2 ◇ ps1)
```

...or must we? This Monoid instance looks a lot like the instance for Dual. In fact, using the `GeneralizedNewtypeDeriving` extension along with Dual, we can define Diagram so that we get the Monoid instance for free again:

```
newtype Diagram = Diagram (Dual [Prim])
  deriving (Semigroup, Monoid)

unD (Diagram (Dual ps)) = ps
prim p                  = Diagram (Dual [p])
mkD ps                  = Diagram (Dual ps)
```

The Monoid instance for Dual [Prim] has exactly the semantics we want; GHC will create a Monoid instance for Diagram from the instance for Dual [Prim] by wrapping and unwrapping Diagram constructors appropriately.

There are drawbacks to this solution, of course: to do anything with Diagram one must now wrap and unwrap both Diagram and Dual constructors. However, there are tools to make this somewhat less tedious (such as the `newtype` package[3]). In any case, the Diagram constructor probably shouldn't be directly exposed to users anyway. The added complexity of using Dual will be hidden in the implementation of a handful of primitive operations on Diagrams.

As for benefits, we have a concise, type-directed specification of the monoidal semantics of Diagram. Some of the responsibility for writing code is shifted onto the compiler, which cuts down on potential sources of error. And although this particular example is simple, working with structurally derived Semigroup and Monoid instances can be an important aid in understanding more complex situations, as we'll see in the next variation.

Variation II: Envelopes

Stacking diagrams via (◇) is a good start, but it's not hard to imagine other modes of composition. For example, consider placing two diagrams "beside" one another, as illustrated in Figure 2.

It is not immediately obvious how this is to be implemented. We evidently need to compute some kind of bounding information for a diagram to decide how it should be positioned relative to others. An idea that first suggests itself is to use *bounding boxes*—that is, axis-aligned rectangles which completely enclose a diagram. However, bounding boxes don't play well with rotation (if you rotate a bounding box by 45 degrees, which bounding box do you

[2] `http://hackage.haskell.org/package/semigroups`

[3] `http://hackage.haskell.org/package/newtype`

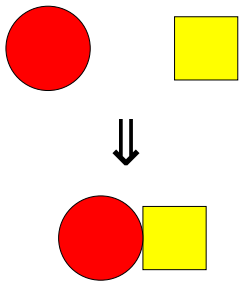

Figure 2. Placing two diagrams beside one another

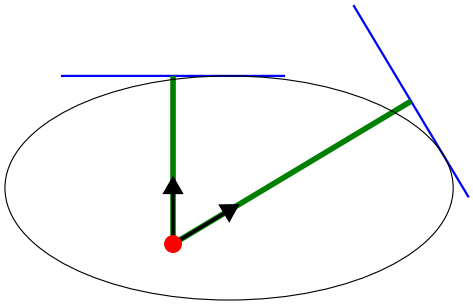

Figure 3. Envelope for an ellipse

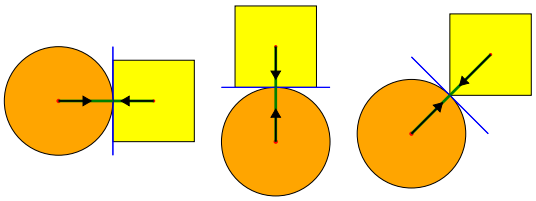

Figure 4. Using envelopes to place diagrams beside one another

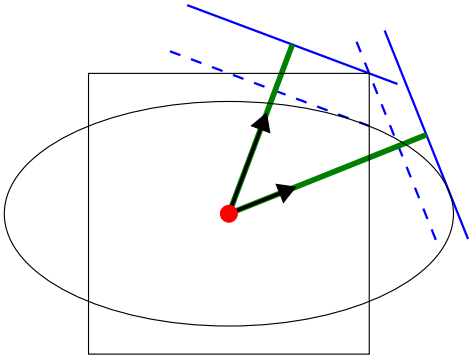

Figure 5. Composing envelopes

get as a result?), and they introduce an inherent left-right-up-down bias—which, though it may be appropriate for something like TEX, is best avoided in a general-purpose drawing library.

An elegant functional solution is something I term an *envelope*.[4] Assume there is a type V_2 representing two-dimensional vectors (and a type P_2 representing points). Then an envelope is a function of type $\mathsf{V}_2 \to \mathbb{R}$.[5] Given a vector v, it returns the minimum distance (expressed as a multiple of v's magnitude) from the origin to a *separating line* perpendicular to v. A separating line is one which partitions space into two half-spaces, one (in the direction opposite v) containing the entirety of the diagram, and the other (in the direction of v) empty. More formally, the envelope yields the smallest real number t such that for every point u inside the diagram, the projection of u (considered as a vector) onto v is equal to some scalar multiple sv with $s \leqslant t$.

Figure 3 illustrates an example. Two query vectors emanate from the origin; the envelope for the ellipse computes the distances to the separating lines shown. Given the envelopes for two diagrams, *beside* can be implemented by querying the envelopes in opposite directions and placing the diagrams on opposite sides of a separating line, as illustrated in Figure 4.

Fundamentally, an envelope represents a convex hull—the locus of all segments with endpoints on a diagram's boundary. However, the term "convex hull" usually conjures up some sort of *intensional* representation, such as a list of vertices. Envelopes, by contrast, are an *extensional* representation of convex hulls; it is only possible to observe examples of their *behavior*.

[4] The initial idea for envelopes is due to Sebastian Setzer. See `http://byorgey.wordpress.com/2009/10/28/collecting-attributes/#comment-2030`.

[5] It might seem cleaner to use *angles* as input to envelopes rather than vectors; however, this definition in terms of vectors generalizes cleanly to higher-dimensional vector spaces, whereas one in terms of angles would not.

Here's the initial definition of Envelope. Assume there is a way to compute an Envelope for any primitive.

```
newtype Envelope = Envelope (V2 → ℝ)
envelopeP :: Prim → Envelope
```

How, now, to compute the Envelope for an entire Diagram? Since *envelopeP* can be used to compute an envelope for each of a diagram's primitives, it makes sense to look for a Monoid structure on envelopes. The envelope for a diagram will then be the combination of the envelopes for all its primitives.

So how do Envelopes compose? If one superimposes a diagram on top of another and then asks for the distance to a separating line in a particular direction, the answer is the *maximum* of the distances for the component diagrams, as illustrated in Figure 5.

Of course, we must check that this operation is associative and has an identity. Instead of trying to check directly, however, let's rewrite the definition of Envelope in a way that makes its compositional semantics apparent, in the same way we did for Diagram using Dual in Variation I.

Since distances are combined with *max*, we can use the Max wrapper defined in `Data.Semigroup`:

```
newtype Envelope = Envelope (V2 → Max ℝ)
  deriving Semigroup
```

The Semigroup instance for Envelope is automatically derived from the instance for Max together with the instance that lifts Semigroup instances over an application of $((\to)\ \mathsf{V}_2)$. The resulting binary operation is exactly the one described above: the input vector is passed as an argument to both envelopes and the results combined using *max*. This also constitutes a proof that the operation is associative, since we already know that Max satisfies the Semigroup law and $((\to)\ \mathsf{V}_2)$ preserves it.

We can now compute the envelope for almost all diagrams: if a diagram contains at least one primitive, apply *envelopeP* to each primitive and then combine the resulting envelopes with $(\diamond)$. We

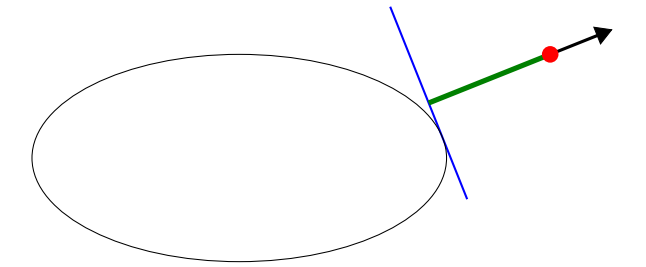

Figure 6. Negative distance as output of an envelope

don't yet know what envelope to assign to the empty diagram, but if Envelope were also an instance of Monoid then we could, of course, use ε.

However, it isn't. The reason has already been explored in the Interlude: there is no smallest real number, and hence no identity element for the reals under *max*. If envelopes actually only returned positive real numbers, we could use $(const\ 0)$ as the identity envelope. However, it makes good sense for an envelope to yield a negative result, if given as input a vector pointing "away from" the diagram; in that case the vector to the separating line is a *negative* multiple of the input vector (see Figure 6).

Since the problem seems to be that there is no smallest real number, the obvious solution is to extend the output type of envelopes to $\mathbb{R} \cup \{-\infty\}$. This would certainly enable a Monoid instance for envelopes; however, it doesn't fit their intended semantics. An envelope must either constantly return $-\infty$ for all inputs (if it corresponds to the empty diagram), or it must return a finite distance for all inputs. Intuitively, if there is "something there" at all, then there is a separating line in every direction, which will have some finite distance from the origin

(It is worth noting that the question of whether diagrams are allowed to have infinite extent in certain directions seems related, but is in fact orthogonal. If this was allowed, envelopes could return $+\infty$ in certain directions, but any valid envelope would still return $-\infty$ for all directions or none.)

So the obvious "solution" doesn't work, but this "all-or-none" aspect of envelopes suggests the correct solution. Simply wrap the entire function type in Option, adding a special distinguished "empty envelope" besides the usual "finite" envelopes implemented as functions. Since Envelope was already an instance of Semigroup, wrapping it in Option will result in a Monoid.

```
newtype Envelope = Envelope (Option (V2 → Max ℝ))
  deriving (Semigroup, Monoid)
```

Looking at this from a slightly different point of view, the most straightforward way to turn a semigroup into a monoid is to use Option; the question is where to insert it. The two potential solutions discussed above are essentially

```
        V2 → Option (Max ℝ)
Option (V2 →        Max ℝ)
```

There is nothing inherently unreasonable about either choice; it comes down to a question of semantics.

In any case, the envelope for any diagram can now be computed using the Monoid instance for Envelope:

```
envelope :: Diagram → Envelope
envelope = hom envelopeP ∘ unD
```

Recall that $hom\ f = mconcat \circ map\ f$ expresses the lifting of a function $a \to m$ to a monoid homomorphism $[a] \to m$.

If we assume that there is a function

```
translateP :: V2 → Prim → Prim
```

to translate any primitive by a given vector, we can concretely implement *beside* as shown below. Essentially, it computes the distance to a separating line for each of the two diagrams (in opposite directions) and translates the second diagram by the sum of the distances before superimposing them. There is a bit of added complication due to handling the possibility that one of the diagrams is empty, in which case the other is returned unchanged (thus making the empty diagram an identity element for *beside*). Note that the $\star$ operator multiplies a vector by a scalar.

```
translate :: V2 → Diagram → Diagram
translate v = mkD ∘ map (translateP v) ∘ unD

unE :: Envelope → Maybe (V2 → ℝ)
unE (Envelope (Option Nothing)) = Nothing
unE (Envelope (Option (Just f))) = Just (getMax ∘ f)

beside :: V2 → Diagram → Diagram → Diagram
beside v d1 d2 =
  case (unE (envelope d1), unE (envelope d2)) of
    (Just e1, Just e2) →
      d1 ◇ translate ((e1 v + e2 (−v)) ⋆ v) d2
    _ →
      d1 ◇ d2
```

Variation III: Caching Envelopes

This method of computing the envelope for a Diagram, while elegant, leaves something to be desired from the standpoint of efficiency. Using *beside* to put two diagrams next to each other requires computing their envelopes. But placing the resulting combined diagram beside something else requires recomputing its envelope from scratch, leading to duplicated work.

In an effort to avoid this, we can try caching the envelope, storing it alongside the primitives. Using the fact that the product of two monoids is a monoid, the compiler can still derive the appropriate instances:

```
newtype Diagram = Diagram (Dual [Prim], Envelope)
  deriving (Semigroup, Monoid)

unD (Diagram (Dual ps, _)) = ps
prim p = Diagram (Dual [p], envelopeP p)
mkD    = hom prim

envelope (Diagram (_, e)) = e
```

Now combining two diagrams with $(\diamond)$ will result in their primitives as well as their cached envelopes being combined. However, it's not *a priori* obvious that this works correctly. We must prove that the cached envelopes "stay in sync" with the primitives—in particular, that if a diagram containing primitives *ps* and envelope *e* has been constructed using only the functions provided above, it satisfies the invariant

$$e = hom\ envelopeP\ ps.$$

Proof. This is true by definition for a diagram constructed with *prim*. It is also true for the empty diagram: since *hom envelopeP* is a monoid homomorphism,

$$hom\ envelopeP\ [\,] = \varepsilon.$$

The interesting case is $(\diamond)$. Suppose we have two diagram values Diagram (Dual ps_1, e_1) and Diagram (Dual ps_2, e_2) for which the invariant holds, and we combine them with $(\diamond)$, resulting in Diagram (Dual $(ps_2 +\!\!+ ps_1), e_1 \diamond e_2$). We must show that the invariant is preserved, that is,

$$e_1 \diamond e_2 = hom\ envelopeP\ (ps_2 +\!\!+ ps_1).$$

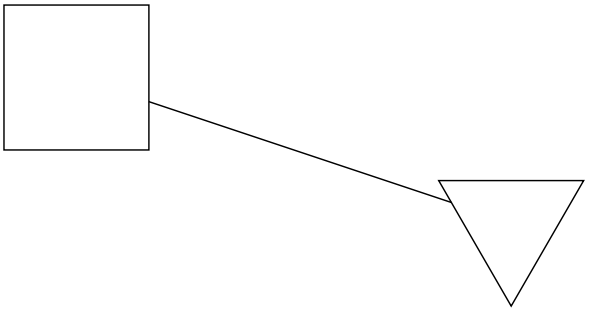

Figure 7. Drawing a line between two shapes

Again, since *hom envelopeP* is a monoid homomorphism,

$$\begin{aligned} &\mathit{hom\ envelopeP}\ (\mathit{ps}_2 +\!\!+ \mathit{ps}_1) \\ &\qquad = \mathit{hom\ envelopeP}\ \mathit{ps}_2 \diamond \mathit{hom\ envelopeP}\ \mathit{ps}_1, \end{aligned}$$

which by assumption is equal to $e_2 \diamond e_1$.

But wait a minute, we wanted $e_1 \diamond e_2$! Never fear: Envelope actually forms a commutative monoid, which can be seen by noting that $\mathsf{Max}\,\mathbb{R}$ is a commutative semigroup, and $((\to)\ \mathsf{V}_2)$ and Option both preserve commutativity. □

Intuitively, it is precisely the fact that the old version of *envelope* (defined in terms of *hom envelopeP*) was a monoid homomorphism which allows caching Envelope values.

Although caching envelopes eliminates some duplicated work, it does not, in and of itself, improve the asymptotic time complexity of something like repeated application of *beside*. Querying the envelope of a diagram with n primitives still requires evaluating $O(n)$ applications of *min*, the same amount of work as constructing the envelope in the first place. However, caching is a prerequisite to *memoizing* envelopes [Michie 1968], which does indeed improve efficiency; the details are omitted in the interest of space.

Variation IV: Traces

Envelopes enable *beside*, but they are not particularly useful for finding actual points on the boundary of a diagram. For example, consider drawing a line between two shapes, as shown in Figure 7. In order to do this, one must compute appropriate endpoints for the line on the boundaries of the shapes, but having their envelopes does not help. As illustrated in Figure 8, envelopes can only give the distance to a separating line, which by definition is a conservative approximation to the actual distance to a diagram's boundary along a given ray.

Consider instead the notion of a *trace*. Given a ray specified by a starting point and a vector giving its direction, the trace computes the distance along the ray to the nearest intersection with a diagram; in other words, it implements a ray/object intersection test just like those used in a ray tracer.

$$\textbf{newtype}\ \mathsf{Trace} = \mathsf{Trace}\ (\mathsf{P}_2 \to \mathsf{V}_2 \to \mathbb{R})$$

The first thing to consider, of course, is how traces combine. Since traces yield the distance to the *nearest* intersection, given two superimposed diagrams, their combined trace should return the *minimum* distance given by their individual traces. We record this declaratively by refining the definition of Trace to

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Trace} = \mathsf{Trace}\ (\mathsf{P}_2 \to \mathsf{V}_2 \to \mathsf{Min}\ \mathbb{R}) \\ &\quad \textbf{deriving}\ (\mathsf{Semigroup}) \end{aligned}$$

Just as with Envelope, this is a semigroup but not a monoid, since there is no largest element of $\mathbb{R}$. Again, inserting Option will make

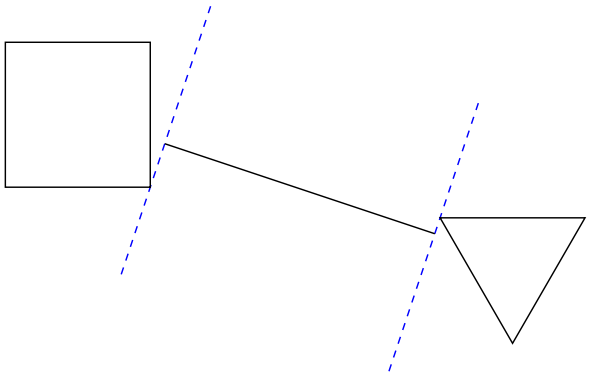

Figure 8. Envelopes are not useful for drawing connecting lines!

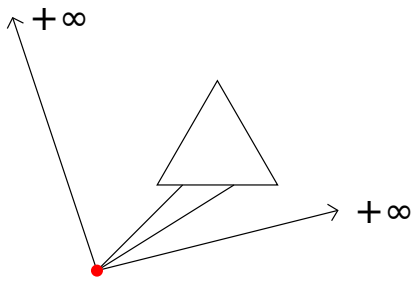

Figure 9. Returning $+\infty$ from a trace

it a monoid; but where should the Option go? It seems there are three possibilities this time (four, if we consider swapping the order of P_2 and V_2):

$$\begin{aligned} &\mathsf{P}_2 \to &&\mathsf{V}_2 \to \mathsf{Option}\ (\mathsf{Min}\ \mathbb{R}) \\ &\mathsf{P}_2 \to \mathsf{Option}\ (\mathsf{V}_2 \to &&\mathsf{Min}\ \mathbb{R}) \\ \mathsf{Option}\ (\mathsf{P}_2 \to &&\mathsf{V}_2 \to &\mathsf{Min}\ \mathbb{R}) \end{aligned}$$

The first represents adjoining $+\infty$ to the output type, and the last represents creating a special, distinguished "empty trace". The second says that there can be certain points from which the diagram is not visible in any direction, while from other points it is not, but this doesn't make sense: if a diagram is visible from any point, then it will be visible everywhere. Swapping P_2 and V_2 doesn't help.

In fact, unlike Envelope, here the first option is best. It is sensible to return $+\infty$ as the result of a trace, indicating that the given ray never intersects the diagram at all (see Figure 9).

Here, then, is the final definition of Trace:

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Trace} = \mathsf{Trace}\ (\mathsf{P}_2 \to \mathsf{V}_2 \to \mathsf{Option}\ (\mathsf{Min}\ \mathbb{R})) \\ &\quad \textbf{deriving}\ (\mathsf{Semigroup}, \mathsf{Monoid}) \end{aligned}$$

Assuming there is a function $\mathit{traceP} :: \mathsf{Prim} \to \mathsf{Trace}$ to compute the trace of any primitive, we could define

$$\begin{aligned} &\mathit{trace} :: \mathsf{Diagram} \to \mathsf{Trace} \\ &\mathit{trace} = \mathit{hom\ traceP} \circ \mathit{unD} \end{aligned}$$

However, this is a monoid homomorphism since Trace is also a commutative monoid, so we can cache the trace of each diagram as well.

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Diagram} \\ &\quad = \mathsf{Diagram}\ (\mathsf{Dual}\ [\mathsf{Prim}], \mathsf{Envelope}, \mathsf{Trace}) \\ &\quad \textbf{deriving}\ (\mathsf{Semigroup}, \mathsf{Monoid}) \end{aligned}$$

Variation V: Transformations and monoid actions

Translation was briefly mentioned in Variation II, but it's time to consider transforming diagrams more generally. Suppose there is a type representing arbitrary *affine transformations*, and a way to apply them to primitives:

data Transformation = ...

transformP :: Transformation → Prim → Prim

Affine transformations include the usual suspects like rotation, reflection, scaling, shearing, and translation; they send parallel lines to parallel lines, but do not necessarily preserve angles. However, the precise definition—along with the precise implementations of Transformation and *transformP*—is not important for our purposes. The important fact, of course, is that Transformation is an instance of Monoid: $t_1 \diamond t_2$ represents the transformation which performs first t_2 and then t_1, and ε is the identity transformation. Given these intuitive semantics, we expect

$$transformP\ \varepsilon\ p = p \tag{1}$$

that is, transforming by the identity transformation has no effect, and

$$transformP\ (t_1 \diamond t_2)\ p = transformP\ t_1\ (transformP\ t_2\ p) \tag{2}$$

that is, $t_1 \diamond t_2$ really does represent doing first t_2 and then t_1. (Equation (2) should make it clear why composition of Transformations is "backwards": for the same reason function composition is "backwards".) Functions satisfying (1) and (2) have a name: *transformP* represents a *monoid action* of Transformation on Prim. Moreover, η-reducing (1) and (2) yields

$$transformP\ \varepsilon = id \tag{1'}$$

$$transformP\ (t_1 \diamond t_2) = transformP\ t_1 \circ transformP\ t_2 \tag{2'}$$

Thus, we can equivalently say that *transformP* is a monoid homomorphism from Transformation to endofunctions on Prim.

Let's make a type class to represent monoid actions:

class Monoid $m \Rightarrow$ Action $m\ a$ **where**
  $act :: m \to a \to a$

instance Action Transformation Prim **where**
  $act = transformP$

(Note that this requires the `MultiParamTypeClasses` extension.) Restating the monoid action laws more generally, for any instance of Action $m\ a$ it should be the case that for all $m_1, m_2 :: m$,

$$act\ \varepsilon = id \tag{MA1}$$

$$act\ (m_1 \diamond m_2) = act\ m_1 \circ act\ m_2 \tag{MA2}$$

When using these laws in proofs we must be careful to note the types at which *act* is applied. Otherwise we might inadvertently use *act* at types for which no instance of Action exists, or—more subtly—circularly apply the laws for the very instance we are attempting to prove lawful. I will use the notation A / B to indicate an appeal to the monoid action laws for the instance Action A B.

Now, consider the problem of applying a transformation to an entire diagram. For the moment, forget about the Dual wrapper and the cached Envelope and Trace, and pretend that a diagram consists solely of a list of primitives. The obvious solution, then, is to map the transformation over the list of primitives.

type Diagram = [Prim]

transformD :: Transformation → Diagram → Diagram
transformD t = *map* (*act* t)

instance Action Transformation Diagram **where**
  $act = transformD$

The Action instance amounts to a claim that *transformD* satisfies the monoid action laws (MA1) and (MA2). The proof makes use of the fact that the list type constructor [] is a functor, that is, $map\ id = id$ and $map\ (f \circ g) = map\ f \circ map\ g$.

Proof.

$$\begin{aligned}
&transformD\ \varepsilon \\
=\ &\{\quad \text{definition of } transformD \quad\} \\
&map\ (act\ \varepsilon) \\
=\ &\{\quad \text{Transformation / Prim} \quad\} \\
&map\ id \\
=\ &\{\quad \text{list functor} \quad\} \\
&id
\end{aligned}$$

$$\begin{aligned}
&transformD\ (t_1 \diamond t_2) \\
=\ &\{\quad \text{definition} \quad\} \\
&map\ (act\ (t_1 \diamond t_2)) \\
=\ &\{\quad \text{Transformation / Prim} \quad\} \\
&map\ (act\ t_1 \circ act\ t_2) \\
=\ &\{\quad \text{list functor} \quad\} \\
&map\ (act\ t_1) \circ map\ (act\ t_2) \\
=\ &\{\quad \text{definition} \quad\} \\
&transformD\ t_1 \circ transformD\ t_2
\end{aligned}$$

□

As an aside, note that this proof actually works for *any* functor, so

instance (Action $m\ a$, Functor f)
  ⇒ Action $m\ (f\ a)$ **where**
  $act\ m = fmap\ (act\ m)$

always defines a lawful monoid action.

Variation VI: Monoid-on-monoid action

The previous variation discussed Transformations and their monoid structure. Recall that Diagram itself is also an instance of Monoid. How does this relate to the action of Transformation? That is, the monoid action laws specify how compositions of transformations act on diagrams, but how do transformations act on compositions of diagrams?

Continuing for the moment to think about the stripped-down variant Diagram = [Prim], we can see first of all that

$$act\ t\ \varepsilon = \varepsilon, \tag{3}$$

since mapping t over the empty list of primitives results in the empty list again. We also have

$$act\ t\ (d_1 \diamond d_2) = (act\ t\ d_1) \diamond (act\ t\ d_2), \tag{4}$$

since

$$\begin{aligned}
&act\ t\ (d_1 \diamond d_2) \\
=\ &\{\quad \text{definitions of } act \text{ and } (\diamond) \quad\} \\
&map\ (act\ t)\ (d_1 +\!\!+ d_2) \\
=\ &\{\quad \text{naturality of } (+\!\!+) \quad\} \\
&map\ (act\ t)\ d_1 +\!\!+ map\ (act\ t)\ d_2 \\
=\ &\{\quad \text{definition} \quad\} \\
&act\ t\ d_1 \diamond act\ t\ d_2
\end{aligned}$$

where the central step follows from a "free theorem" [Wadler 1989] derived from the type of $(+\!\!+)$.

Equations (3) and (4) together say that the action of any particular Transformation is a monoid homomorphism from Diagram

to itself. This sounds desirable: when the type being acted upon has some structure, we want the monoid action to preserve it. From now on, we include these among the monoid action laws when the type being acted upon is also a Monoid:

$$act\ m\ \varepsilon = \varepsilon \tag{MA3}$$

$$act\ m\ (n_1 \diamond n_2) = act\ m\ n_1 \diamond act\ m\ n_2 \tag{MA4}$$

It's finally time to stop pretending: so far, a value of type Diagram contains not only a (dualized) list of primitives, but also cached Envelope and Trace values. When applying a transformation to a Diagram, something must be done with these cached values as well. An obviously correct but highly unsatisfying approach would be to simply throw them away and recompute them from the transformed primitives every time.

However, there is a better way: all that's needed is to define an action of Transformation on both Envelope and Trace, subject to (MA1)–(MA4) along with

$$act\ t \circ envelopeP = envelopeP \circ act\ t \tag{TE}$$

$$act\ t \circ traceP = traceP \circ act\ t \tag{TT}$$

Equations (TE) and (TT) specify that transforming a primitive's envelope (or trace) should be the same as first transforming the primitive and then finding the envelope (respectively trace) of the result. (Intuitively, it would be quite strange if these did *not* hold; we could even take them as the *definition* of what it means to transform a primitive's envelope or trace.)

```
instance Action Transformation Envelope where
  ...
instance Action Transformation Trace where
  ...
instance Action Transformation Diagram where
  act t (Diagram (Dual ps, e, tr))
    = Diagram (Dual (map (act t) ps), act t e, act t tr)
```

Incidentally, it is not *a priori* obvious that such instances can even be defined—the action of Transformation on Envelope in particular is nontrivial and quite interesting. However, it is beyond the scope of this paper.

We must prove that this gives the same result as throwing away the cached Envelope and Trace and then recomputing them directly from the transformed primitives. The proof for Envelope is shown here; the proof for Trace is entirely analogous.

As established in Variation III, the envelope e stored along with primitives ps satisfies the invariant

$$e = hom\ envelopeP\ ps.$$

We must therefore prove that

$$act\ t\ (hom\ envelopeP\ ps) = hom\ envelopeP\ (map\ (act\ t)\ ps),$$

or, in point-free form,

$$act\ t \circ hom\ envelopeP = hom\ envelopeP \circ map\ (act\ t).$$

Proof. We reason as follows:

$$\begin{aligned}
&act\ t \circ hom\ envelope\ \mathsf{P} \\
=\ &\{\quad \text{definition} \quad\} \\
&act\ t \circ mconcat \circ map\ envelopeP \\
=\ &\{\quad \text{lemma proved below} \quad\} \\
&mconcat \circ map\ (act\ t) \circ map\ envelopeP \\
=\ &\{\quad \text{list functor, (TE)} \quad\} \\
&mconcat \circ map\ envelopeP \circ map\ (act\ t) \\
=\ &\{\quad \text{definition} \quad\} \\
&hom\ envelopeP \circ map\ (act\ t)
\end{aligned}$$

□

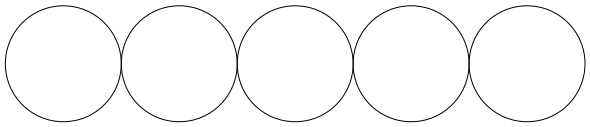

Figure 10. Laying out a line of circles with *beside*

It remains only to prove that $act\ t \circ mconcat = mconcat \circ map\ (act\ t)$. This is where the additional monoid action laws (MA3) and (MA4) come in. The proof also requires some standard facts about *mconcat*, which are proved in the Appendix.

Proof. The proof is by induction on an arbitrary list (call it l) given as an argument to $act\ t \circ mconcat$. If l is the empty list,

$$\begin{aligned}
&act\ t\ (mconcat\ [\,]) \\
=\ &\{\quad mconcat \quad\} \\
&act\ t\ \varepsilon \\
=\ &\{\quad \text{monoid action (MA3)} \quad\} \\
&\varepsilon \\
=\ &\{\quad mconcat\text{, definition of } map \quad\} \\
&mconcat\ (map\ (act\ t)\ [\,])
\end{aligned}$$

In the case that $l = x : xs$,

$$\begin{aligned}
&act\ t\ (mconcat\ (x : xs)) \\
=\ &\{\quad mconcat \quad\} \\
&act\ t\ (x \diamond mconcat\ xs) \\
=\ &\{\quad \text{monoid action (MA4)} \quad\} \\
&act\ t\ x \diamond act\ t\ (mconcat\ xs) \\
=\ &\{\quad \text{induction hypothesis} \quad\} \\
&act\ t\ x \diamond mconcat\ (map\ (act\ t)\ xs) \\
=\ &\{\quad mconcat \quad\} \\
&mconcat\ (act\ t\ x : map\ (act\ t)\ xs) \\
=\ &\{\quad \text{definition of } map \quad\} \\
&mconcat\ (map\ (act\ t)\ (x : xs))
\end{aligned}$$

□

Variation VII: Efficiency via deep embedding

Despite the efforts of the previous variation, applying transformations to diagrams is still not as efficient as it could be. The problem is that applying a transformation always requires a full traversal of the list of primitives. To see why this is undesirable, imagine a scenario where we alternately superimpose a new primitive on a diagram, transform the result, add another primitive, transform the result, and so on. In fact, this is exactly what happens when using *beside* repeatedly to lay out a line of diagrams, as in the following code (whose result is shown in Figure 10):

```
unit_x :: V2   — unit vector along the positive x-axis
hcat = foldr (beside unit_x) ε
lineOfCircles n = hcat (replicate n circle)
```

Fully evaluating *lineOfCircles n* takes $O(n^2)$ time, because the kth call to *beside* must map over k primitives, resulting in $1+2+3+\cdots+n$ total calls to *transformP*. (Another problem is that it results in left-nested calls to $(+\!\!+)$; this is dealt with in the next variation.) Can this be improved?

Consider again the monoid action law

$$act\ (t_1 \diamond t_2) = act\ t_1 \circ act\ t_2.$$

Read from right to left, it says that instead of applying two transformations (resulting in two traversals of the primitives), one can

achieve the same effect by first combining the transformations and then doing a single traversal. Taking advantage of this requires some way to delay evaluation of transformations until the results are demanded, and a way to collapse multiple delayed transformations before actually applying them.

A first idea is to store a "pending" transformation along with each diagram:

```
newtype Diagram =
  Diagram (Dual [Prim], Transformation, Envelope, Trace)
```

In order to apply a new transformation to a diagram, simply combine it with the stored one:

```
instance Action Transformation Diagram where
  act t' (Diagram (ps, t, e, tr))
    = Diagram (ps, t' ◇ t, act t' e, act t' tr)
```

However, we can no longer automatically derive Semigroup or Monoid instances for Diagram—that is to say, we *could*, but the semantics would be wrong! When superimposing two diagrams, it does not make sense to combine their pending transformations. Instead, the transformations must be applied before combining:

```
instance Semigroup Diagram where
  (Diagram (ps1, t1, e1, tr1)) ◇ (Diagram (ps2, t2, e2, tr2))
    = Diagram (act t1 ps1 ◇ act t2 ps2,
               ε,
               e1 ◇ e2,
               tr1 ◇ tr2)
```

So, transformations are delayed somewhat—but only until a call to $(\diamond)$, which forces them to be applied. This helps with consecutive transformations, but doesn't help at all with the motivating scenario from the beginning of this variation, where transformations are interleaved with compositions.

In order to really make a difference, this idea of delaying transformations must be taken further. Instead of being delayed only until the next composition, they must be delayed as long as possible, until forced by an *observation*. This, in turn, forces a radical redesign of the Diagram structure. In order to delay interleaved transformations and compositions, a tree structure is needed—though a Diagram will still be a list of primitives from a semantic point of view, an actual list of primitives no longer suffices as a concrete representation.

The key to designing an appropriate tree structure is to think of the functions that create diagrams as an *algebraic signature*, and construct a data type corresponding to the *free algebra* over this signature [Turner 1985]. Put another way, so far we have a *shallow embedding* of a domain-specific language for constructing diagrams, where the operations are carried out immediately on semantic values, but we need a *deep embedding*, where operations are first reified into an abstract syntax tree and interpreted later.

More concretely, here are the functions we've seen so far with a result type of Diagram:

```
prim :: Prim → Diagram
ε    :: Diagram
(◇)  :: Diagram → Diagram → Diagram
act  :: Transformation → Diagram → Diagram
```

We simply make each of these functions into a data constructor, remembering to also cache the envelope and trace at every node corresponding to $(\diamond)$:

```
data Diagram
  = Prim Prim
  | Empty
  | Compose (Envelope, Trace) Diagram Diagram
  | Act Transformation Diagram
```

There are a few accompanying functions and instances to define. First, to extract the Envelope of a Diagram, just do the obvious thing for each constructor (extracting the Trace is analogous):

```
envelope :: Diagram →                Envelope
envelope (Prim p)                    = envelopeP p
envelope Empty                       = ε
envelope (Compose (e, _) _ _)        = e
envelope (Act t d)                   = act t (envelope d)
```

By this point, there is certainly no way to automatically derive Semigroup and Monoid instances for Diagram, but writing them manually is not complicated. Empty is explicitly treated as the identity element, and composition is delayed with the Compose constructor, extracting the envelope and trace of each subdiagram and caching their compositions:

```
instance Semigroup Diagram where
  Empty ◇ d = d
  d ◇ Empty = d
  d1 ◇ d2
    = Compose
        (envelope d1 ◇ envelope d2
        , trace      d1 ◇ trace      d2
        )
        d1 d2
instance Monoid Diagram where
  ε = Empty
```

The particularly attentive reader may have noticed something strange about this Semigroup instance: $(\diamond)$ is not associative! $d_1 \diamond (d_2 \diamond d_3)$ and $(d_1 \diamond d_2) \diamond d_3$ are not equal, since they result in trees of two different shapes. However, intuitively it seems that $d_1 \diamond (d_2 \diamond d_3)$ and $(d_1 \diamond d_2) \diamond d_3$ are still "morally" the same, that is, they are two representations of "the same" diagram. We can formalize this idea by considering Diagram as a *quotient* type, using some equivalence relation other than structural equality. In particular, associativity does hold if we consider two diagrams d_1 and d_2 equivalent whenever $unD\ d_1 \equiv unD\ d_2$, where $unD :: \mathsf{Diagram} \to [\mathsf{Prim}]$ "compiles" a Diagram into a flat list of primitives. The proof is omitted; given the definition of *unD* below, it is straightforward and unenlightening.

The action of Transformation on the new version of Diagram can be defined as follows:

```
instance Action Transformation Diagram where
  act t Empty      = Empty
  act t (Act t' d) = Act (t ◇ t') d
  act t d          = Act t d
```

Although the monoid action laws (MA1) and (MA2) hold by definition, (MA3) and (MA4) again hold only up to semantic equivalence (the proof is similarly straightforward).

Finally, we define *unD*, which "compiles" a Diagram into a flat list of Prims. A simple first attempt is just an interpreter that replaces each constructor by the operation it represents:

```
unD :: Diagram →                [Prim]
unD (Prim p)                    = [p]
unD Empty                       = ε
unD (Compose _ d1 d2)           = unD d2 ◇ unD d1
unD (Act t d)                   = act t (unD d)
```

This seems obviously correct, but brings us back exactly where we started: the whole point of the new tree-like Diagram type was to improve efficiency, but so far we have only succeeded in pushing work around! The benefit of having a deep embedding is that we can do better than a simple interpreter, by doing some sort of nontrivial analysis of the expression trees.

In this particular case, all we need to do is pass along an extra parameter accumulating the "current transformation" as we recurse down the tree. Instead of immediately applying each transformation as it is encountered, we simply accumulate transformations as we recurse and apply them when reaching the leaves. Each primitive is processed exactly once.

```
unD' :: Diagram → [Prim]
unD' = go ε where
  go :: Transformation → Diagram → [Prim]
  go t (Prim p)              = [act t p]
  go _ Empty                 = ε
  go t (Compose _ d1 d2)     = go t d2 ◇ go t d1
  go t (Act t' d)            = go (t ◇ t') d
```

Of course, we ought to prove that *unD* and *unD′* yield identical results—as it turns out, the proof makes use of all four monoid action laws. To get the induction to go through requires proving the stronger result that for all transformations t and diagrams d,

$$act\ t\ (unD\ d) = go\ t\ d.$$

From this it will follow, by (MA1), that

$$unD\ d = act\ \varepsilon\ (unD\ d) = go\ \varepsilon\ d = unD'\ d.$$

Proof. By induction on d.

- If $d = \mathsf{Prim}\ p$, then $act\ t\ (unD\ (\mathsf{Prim}\ p)) = act\ t\ [p] = [act\ t\ p] = go\ t\ (\mathsf{Prim}\ p)$.
- If $d = \mathsf{Empty}$, then $act\ t\ (unD\ \mathsf{Empty}) = act\ t\ \varepsilon = \varepsilon = go\ t\ \mathsf{Empty}$, where the central equality is (MA3).
- If $d = \mathsf{Compose}\ c\ d_1\ d_2$, then

  $act\ t\ (unD\ (\mathsf{Compose}\ c\ d_1\ d_2))$
  $= \{$ definition $\}$
  $act\ t\ (unD\ d_2 \diamond unD\ d_1)$
  $= \{$ monoid action (MA4) $\}$
  $act\ t\ (unD\ d_2) \diamond act\ t\ (unD\ d_1)$
  $= \{$ induction hypothesis $\}$
  $go\ t\ d_2 \diamond go\ t\ d_1$
  $= \{$ definition $\}$
  $go\ t\ (\mathsf{Compose}\ c\ d_1\ d_2)$

- Finally, if $d = \mathsf{Act}\ t'\ d'$, then

  $act\ t\ (unD\ (\mathsf{Act}\ t'\ d'))$
  $= \{$ definition $\}$
  $act\ t\ (act\ t'\ (unD\ d'))$
  $= \{$ monoid action (MA2) $\}$
  $act\ (t \diamond t')\ (unD\ d')$
  $= \{$ induction hypothesis $\}$
  $go\ (t \diamond t')\ d'$
  $= \{$ definition $\}$
  $go\ t\ (\mathsf{Act}\ t'\ d')$

□

Variation VIII: Difference lists

Actually, *unD′* still suffers from another performance problem hinted at in the previous variation. A right-nested expression like $d_1 \diamond (d_2 \diamond (d_3 \diamond d_4))$ still takes quadratic time to compile, because it results in left-nested calls to $(+\!\!+)$. This can be solved using *difference lists* [Hughes 1986]: the idea is to represent a list $xs :: [a]$ using the function $(xs+\!\!+) :: [a] \to [a]$. Appending two lists is then accomplished by composing their functional representations. The "trick" is that left-nested function composition ultimately results in reassociated (right-nested) appends:

$$(((xs+\!\!+) \circ (ys+\!\!+)) \circ (zs+\!\!+))\ [\,] = xs +\!\!+ (ys +\!\!+ (zs +\!\!+ [\,])).$$

In fact, difference lists arise from viewing

$$(+\!\!+) :: [a] \to ([a] \to [a])$$

itself as a monoid homomorphism, from the list monoid to the monoid of endomorphisms on $[a]$. (H1) states that $(+\!\!+)\ \varepsilon = \varepsilon$, which expands to $(+\!\!+)\ [\,] = id$, that is, $[\,] +\!\!+ xs = xs$, which is true by definition. (H2) states that $(+\!\!+)\ (xs \diamond ys) = (+\!\!+)\ xs \diamond (+\!\!+)\ ys$, which can be rewritten as

$$((xs +\!\!+ ys)+\!\!+) = (xs+\!\!+) \circ (ys+\!\!+).$$

In this form, it expresses that function composition is the correct implementation of append for difference lists. Expand it a bit further by applying both sides to an arbitrary argument zs,

$$(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$$

and it resolves itself into the familiar associativity of $(+\!\!+)$.

Here, then, is a yet further improved variant of *unD*:

```
unD'' :: Diagram → [Prim]
unD'' d = appEndo (go ε d) [] where
  go :: Transformation → Diagram → Endo [Prim]
  go t (Prim p)              = Endo ((act t p):)
  go _ Empty                 = ε
  go t (Compose _ d1 d2)     = go t d2 ◇ go t d1
  go t (Act t' d)            = go (t ◇ t') d
```

Variation IX: Generic monoidal trees

Despite appearances, there is nothing really specific to diagrams about the structure of the Diagram data type. There is one constructor for "leaves", two constructors representing a monoid structure, and one representing monoid actions. This suggests generalizing to a polymorphic type of "monoidal trees":

```
data MTree d u l
  = Leaf u l
  | Empty
  | Compose u (MTree d u l) (MTree d u l)
  | Act d (MTree d u l)
```

d represents a "downwards-traveling" monoid, which acts on the structure and accumulates along paths from the root. u represents an "upwards-traveling" monoid, which originates in the leaves and is cached at internal nodes. l represents the primitive data which is stored in the leaves.

We can now redefine Diagram in terms of MTree:

```
type Diagram
  = MTree Transformation (Envelope, Trace) Prim
prim p = Leaf (envelopeP p, traceP p) p
```

There are two main differences between MTree and Diagram. First, the pair of monoids, Envelope and Trace, have been replaced by a single u parameter—but since a pair of monoids is again a monoid, this is really not a big difference after all. All that is needed is an instance for monoid actions on pairs:

```
instance (Action m a, Action m b)
  ⇒ Action m (a,b) where
  act m (a,b) = (act m a, act m b)
```

The proof of the monoid action laws for this instance is left as a straightforward exercise.

A second, bigger difference is that the Leaf constructor actually stores a value of type u along with the value of type l, whereas

the Prim constructor of Diagram stored only a Prim. Diagram could get away with this because the specific functions *envelopeP* and *traceP* were available to compute the Envelope and Trace for a Prim when needed. In the general case, some function of type $(l \to u)$ would have to be explicitly provided to MTree operations —instead, it is cleaner and easier to cache the result of such a function at the time a Leaf node is created.

Extracting the u value from an MTree is thus straightforward. This generalizes both *envelope* and *trace*:

```
getU :: (Action d u, Monoid u) ⇒ MTree d u l → u
getU (Leaf u _)          = u
getU Empty               = ε
getU (Compose u _ _) = u
getU (Act d t)           = act d (getU t)
envelope = fst ∘ getU
trace    = snd ∘ getU
```

The Semigroup and Action instances are straightforward generalizations of the instances from Variation VII.

```
instance (Action d u, Monoid u)
    ⇒ Semigroup (MTree d u l) where
  Empty ◇ t = t
  t ◇ Empty = t
  t1 ◇ t2   = Compose (getU t1 ◇ getU t2) t1 t2
instance Semigroup d ⇒ Action d (MTree d u l) where
  act _ Empty    = Empty
  act d (Act d' t) = Act (d ◇ d') t
  act d t          = Act d t
```

In place of *unD*, we define a generic fold for MTree, returning not a list but an arbitrary monoid. There's really not much difference between returning an arbitrary monoid and a free one (*i.e.* a list), but it's worth pointing out that the idea of "difference lists" generalizes to arbitrary "difference monoids": $(\diamond)$ itself is a monoid homomorphism.

```
foldMTree :: (Monoid d, Monoid r, Action d r)
          ⇒ (l → r) → MTree d u l → r
foldMTree leaf t = appEndo (go ε t) ε where
  go d (Leaf _ l)          = Endo (act d (leaf l) ◇)
  go _ Empty               = ε
  go d (Compose _ t1 t2) = go d t1 ◇ go d t2
  go d (Act d' t)          = go (d ◇ d') t
unD :: Diagram → [Prim]
unD = getDual ∘ foldMTree (Dual ∘ (:[]))
```

Again, associativity of $(\diamond)$ and the monoid action laws only hold up to semantic equivalence, defined in terms of *foldMTree*.

Variation X: Attributes and product actions

So far, there's been no mention of fill color, stroke color, transparency, or other similar *attributes* we might expect diagrams to possess. Suppose there is a type Style representing collections of attributes. For example, {Fill Purple, Stroke Red} :: Style might indicate a diagram drawn in red and filled with purple. Style is then an instance of Monoid, with ε corresponding to the Style containing no attributes, and $(\diamond)$ corresponding to right-biased union. For example,

```
{Fill Purple, Stroke Red} ◇ {Stroke Green, Alpha 0.3}
                  = {Fill Purple, Stroke Green, Alpha 0.3}
```

where Stroke Green overrides Stroke Red. We would also expect to have a function

```
applyStyle :: Style → Diagram → Diagram
```

for applying a Style to a Diagram. Of course, this sounds a lot like a monoid action! However, it is not so obvious how to implement a new monoid action on Diagram. The fact that Transformation has an action on Diagram is encoded into its definition, since the first parameter of MTree is a "downwards" monoid with an action on the structure:

```
type Diagram
  = MTree Transformation (Envelope, Trace) Prim
```

Can we simply replace Transformation with the product monoid (Transformation, Style)? Instances for Action Style Envelope and Action Style Trace need to be defined, but these can just be trivial, since styles presumably have no effect on envelopes or traces:

```
instance Action Style Envelope where
  act _ = id
```

In fact, the only other thing missing is an Action instance defining the action of a product monoid. One obvious instance is:

```
instance (Action m1 a, Action m2 a)
    ⇒ Action (m1, m2) a where
  act (m1, m2) = act m1 ∘ act m2
```

though it's not immediately clear whether this satisfies the monoid action laws. It turns out that (MA1), (MA3), and (MA4) do hold and are left as exercises. However, (MA2) is a bit more interesting. It states that we should have

$$act\ ((m_{11}, m_{21}) \diamond (m_{12}, m_{22})) = act\ (m_{11}, m_{21}) \circ act\ (m_{12}, m_{22}).$$

Beginning with the left-hand side,

$$\begin{aligned} & act\ ((m_{11}, m_{21}) \diamond (m_{12}, m_{22})) \\ = \ & \{ \quad \text{product monoid} \quad \} \\ & act\ (m_{11} \diamond m_{12}, m_{21} \diamond m_{22}) \\ = \ & \{ \quad \text{proposed definition of } act \text{ for pairs} \quad \} \\ & act\ (m_{11} \diamond m_{12}) \circ act\ (m_{21} \diamond m_{22}) \\ = \ & \{ \quad m_1 \,/\, a, m_2 \,/\, a \ \text{(MA2)} \quad \} \\ & act\ m_{11} \circ act\ m_{12} \circ act\ m_{21} \circ act\ m_{22} \end{aligned}$$

But the right-hand side yields

$$\begin{aligned} & act\ (m_{11}, m_{21}) \circ act\ (m_{12}, m_{22}) \\ = \ & \{ \quad \text{proposed definition of } act \quad \} \\ & act\ m_{11} \circ act\ m_{21} \circ act\ m_{12} \circ act\ m_{22} \end{aligned}$$

In general, these will be equal only when $act\ m_{12} \circ act\ m_{21} = act\ m_{21} \circ act\ m_{12}$—and since these are all arbitrary elements of the types m_1 and m_2, (MA2) will hold precisely when the actions of m_1 and m_2 commute. Intuitively, the problem is that the product of two monoids represents their "parallel composition", but defining the action of a pair requires arbitrarily picking one of the two possible orders for the elements to act. The monoid action laws hold precisely when this arbitrary choice of order makes no difference.

Ultimately, if the action of Transformation on Diagram commutes with that of Style—which seems reasonable—then adding attributes to diagrams essentially boils down to defining

```
type Diagram = MTree (Transformation, Style)
                     (Envelope, Trace)
                     Prim
```

Coda

Monoid homomorphisms have been studied extensively in the program derivation community, under the slightly more general framework of *list homomorphisms* [Bird 1987]. Much of the presentation

here involving monoid homomorphisms can be seen as a particular instantiation of that work.

There is much more that can be said about monoids as they relate to library design. There is an intimate connection between monoids and Applicative functors, which indeed are also known as *monoidal* functors. Parallel to Semigroup is a variant of Applicative lacking the *pure* method, which also deserves more attention. Monads are (infamously) monoidal in a different sense. More fundamentally, categories are "monoids with types".

Beyond monoids, the larger point is that library design should be driven by elegant underlying mathematical structures, and especially by homomorphisms [Elliott 2009].

Acknowledgments

Thanks to Daniel Wagner and Vilhelm Sjöberg for being willing to listen to my ramblings about diagrams and for offering many helpful insights over the years. I'm also thankful to the regulars in the `#diagrams` IRC channel (Drew Day, Claude Heiland-Allen, Deepak Jois, Michael Sloan, Luite Stegeman, Ryan Yates, and others) for many helpful suggestions, and simply for making diagrams so much fun to work on. A big thank you is also due Conal Elliott for inspiring me to think more deeply about semantics and homomorphisms, and for providing invaluable feedback on a very early version of diagrams. Finally, I'm grateful to the members of the Penn PLClub for helpful feedback on an early draft of this paper, and to the anonymous reviewers for a great many helpful suggestions.

This material is based upon work supported by the National Science Foundation under Grant Nos. 1116620 and 1218002.

Appendix

Given the definition $mconcat = foldr\ (\diamond)\ \varepsilon$, we compute $mconcat\ [\,] = foldr\ (\diamond)\ \varepsilon\ [\,] = \varepsilon$, and

$$\begin{aligned} & mconcat\ (x : xs) \\ &= foldr\ (\diamond)\ \varepsilon\ (x : xs) \\ &= x \diamond foldr\ (\diamond)\ \varepsilon\ xs \\ &= x \diamond mconcat\ xs. \end{aligned}$$

These facts are referenced in proof justification steps by the hint *mconcat*.

Next, recall the definition of *hom*, namely

$$\begin{aligned} & hom :: \mathsf{Monoid}\ m \Rightarrow (a \to m) \to ([a] \to m) \\ & hom\ f = mconcat \circ map\ f \end{aligned}$$

We first note that

$$\begin{aligned} & hom\ f\ (x : xs) \\ = \ & \{ \quad \text{definition of } hom \text{ and } map \quad \} \\ & mconcat\ (f\ x : map\ f\ xs) \\ = \ & \{ \quad mconcat \quad \} \\ & f\ x \diamond mconcat\ (map\ f\ xs) \\ = \ & \{ \quad \text{definition of } hom \quad \} \\ & f\ x \diamond hom\ f\ xs \end{aligned}$$

We now prove that $hom\ f$ is a monoid homomorphism for all f.

Proof. First, $hom\ f\ [\,] = (mconcat \circ map\ f)\ [\,] = mconcat\ [\,] = \varepsilon$ (H1).

Second, we show (H2), namely,

$$hom\ f\ (xs +\!\!+ ys) = hom\ f\ xs \diamond hom\ f\ ys,$$

by induction on xs.

- If $xs = [\,]$, we have $hom\ f\ ([\,] +\!\!+ ys) = hom\ f\ ys = \varepsilon \diamond hom\ f\ ys = hom\ f\ [\,] \diamond hom\ f\ ys$.
- Next, suppose $xs = x : xs'$:

$$\begin{aligned} & hom\ f\ ((x : xs') +\!\!+ ys) \\ & \qquad \{ \quad \text{definition of } (+\!\!+) \quad \} \\ = \ & hom\ f\ (x : (xs' +\!\!+ ys)) \\ & \qquad \{ \quad hom \text{ of } (:), \text{ proved above} \quad \} \\ = \ & f\ x \diamond hom\ f\ (xs' +\!\!+ ys) \\ & \qquad \{ \quad \text{induction hypothesis} \quad \} \\ = \ & f\ x \diamond hom\ f\ xs' \diamond hom\ f\ ys \\ & \qquad \{ \quad \text{associativity of } (\diamond) \text{ and } hom \text{ of } (:) \quad \} \\ = \ & (hom\ f\ (x : xs')) \diamond hom\ f\ ys \end{aligned}$$

□

As a corollary, $mconcat\ (xs +\!\!+ ys) = mconcat\ xs +\!\!+ mconcat\ ys$, since $hom\ id = mconcat \circ map\ id = mconcat$.