Monoids: Theme and Variations (Functional Pearl)

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

Follow this and additional works at: http://repository.upenn.edu/cis_papers
Part of the Programming Languages and Compilers Commons, Software Engineering Commons, and the Theory and Algorithms Commons

Recommended Citation
Brent A. Yorgey, "Monoids: Theme and Variations (Functional Pearl)", July 2012.
Monoids: Theme and Variations (Functional Pearl)

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.

Keywords
monoid, homomorphism, monoid action, EDSL

Disciplines
Programming Languages and Compilers | Software Engineering | Theory and Algorithms

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/762
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 Tech-
niques]: 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,

1http://projects.haskell.org/diagrams/

Figure 1. Superimposing a list of primitives

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 “bot-
tommost” and ending with the “topmost” (see Figure 1).

Lists support concatenation, and “concatenating” two Diagrams
also makes good sense: concatenation of lists of primitives corre-
sponds to superposition of diagrams—that is, placing one diagram
on top of another. The empty list is an identity element for con-
catenation ([] ++ 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:


- 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 \( \cdot : S \to S \to S \) and a distinguished element \( \varepsilon : S \), subject to the laws

\[
\begin{align*}
\varepsilon \cdot x &= x = x \cdot \varepsilon \quad &\text{(M1)} \\
\varepsilon \cdot (y \cdot z) &= (x \cdot y) \cdot z. &\text{(M2)}
\end{align*}
\]

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

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

```haskell
class Monoid a where
  \varepsilon :: a
  (\cdot) :: a \to a \to a
  mconcat :: [a] \to a
  mconcat = foldr (\cdot) \varepsilon
```

The actual `Monoid` methods are named `mempty` and `mappend`, but I will use \( \varepsilon \) and \( (\cdot) \) in the interest of brevity.

`mconcat` “reduces” a list using \( (\cdot) \), that is,

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

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 \( \mathbb{Z}, \mathbb{Q}, \mathbb{R} \) 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:

\[
\begin{align*}
\varepsilon &= \text{tuple} (\varepsilon, \varepsilon) \\
(m_1, n_1) \cdot (m_2, n_2) &= (m_1 \cdot m_2, n_1 \cdot n_2)
\end{align*}
\]

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

\[
\begin{align*}
\text{instance } \text{Monoid } m \Rightarrow \text{Monoid } (a \to m) &\text{ where} \\
\varepsilon &= \text{const } \varepsilon \\
f_1 \cdot f_2 &= \lambda a \to f_1 \circ f_2 \ a
\end{align*}
\]

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:

\[
\text{newtype } \text{Dual } a = \text{Dual } a
\]

\[
\text{instance } \text{Monoid } a \Rightarrow \text{Monoid } (\text{Dual } a) &\text{ where} \\
\varepsilon &= \text{Dual } \varepsilon \\
(f_1 \cdot f_2) &= \lambda a \to (f_1 \circ f_2) \ a
\]

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

\[ x \cdot y = y \cdot x \]

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 \( (\cdot), ((\cdot) \varepsilon) \), 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

\[
\begin{align*}
f \varepsilon &= \varepsilon &\text{(H1)} \\
f (x \cdot y) &= f x \cdot f y &\text{(H2)}
\end{align*}
\]

For example, \( \text{length } \ [\] = 0 \) and \( \text{length } (xs ++ ys) = \text{length } xs + \text{length } ys \), 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:

\[
\text{hom} : \text{Monoid } m \Rightarrow (a \to m) \to ([a] \to m)
\]

\[
\text{hom } f = mconcat \circ \text{map } f
\]

See the Appendix for a proof that \text{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:

```haskell
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 \(\text{Min}\) as

```haskell
newtype Min a = Min { getMin :: a }
instance Ord a => Semigroup (Min a) where
    Min a □ Min b = Min (min a b)
```

and \(\text{Max}\) 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:

```haskell
instance Monoid a => Monoid (Maybe a) where
    empty = Nothing
    Nothing □ Nothing = Nothing
    (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:

```haskell
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 \(d_1 □ d_2\) 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 \(d_1 □ d_2\) will result in \(d_1\) being on top. First, we must wrap \([\text{Prim}]\) in a newtype. We also define a few helper functions for dealing with the newtype constructor:

```haskell
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:

```haskell
instance Monoid Diagram where
    empty = 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:

```haskell
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 \([\text{Prim}]\) has exactly the semantics we want; GHC will create a Monoid instance for Diagram from the instance for Dual \([\text{Prim}]\) 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\(^1\)). 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 \((\circ)\) 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

\(\text{1http://hackage.haskell.org/package/newtype}\)
gram, the projection of the smallest real number opposite which partitions space into two half-spaces, one (in the direction of $v$) 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 $s \leq t$.

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.

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

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

```haskell
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`:

```haskell
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 `((→) V2)`. 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 `((→) V2)` 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 `⋄`. We
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 \( \epsilon \).

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 \( \langle \text{const } 0 \rangle \) as the identity envelope. However, it makes good sense for an envelope to yield a negative distance from the origin, 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.

\[
\text{newtype Envelope} = \text{Envelope } (\text{Option } (V \to \text{Max } \mathbb{R}))
\]
\[
\text{deriving } (\text{Semigroup, Monoid})
\]

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

\[
V \to \text{Option } (\text{Max } \mathbb{R})
\]
\[
\text{Option } (V \to \text{Max } \mathbb{R})
\]

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:

\[
\text{envelope} :: \text{Diagram } \to \text{Envelope}
\]
\[
\text{envelope} = \text{hom envelopeP } \circ \text{unD}
\]

Recall that \( \text{hom } f = \text{mconcat } \circ \text{map } f \) expresses the lifting of a function \( a \to m \) to a monoid homomorphism \( [a] \to m \).

If we assume that there is a function \( \text{translateP} :: V \to \text{Prim } \to \text{Prim} \) to translate any primitive by a given vector, we can concretely implement \( \text{beside} \) 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 \( \text{beside} \)). Note that the \( \ast \) operator multiplies a vector by a scalar.

\[
\text{translate} :: V \to \text{Diagram } \to \text{Diagram}
\]
\[
\text{translate } v = \text{mkD } \circ \text{map } (\text{translateP } v) \circ \text{unD}
\]
\[
\text{unE} :: \text{Envelope } \to \text{Maybe } (V \to \mathbb{R})
\]
\[
\text{unE } (\text{Envelope } (\text{Option Nothing})) = \text{Nothing}
\]
\[
\text{unE } (\text{Envelope } (\text{Option } (\text{Just } f))) = \text{Just } (\text{getMax } \circ f)
\]
\[
\text{beside} :: V \to \text{Diagram } \to \text{Diagram } \to \text{Diagram}
\]
\[
\text{beside } v d_1 d_2 =
\]
\[
\text{case } (\text{unE } (\text{envelope } d_1), \text{unE } (\text{envelope } d_2)) \text{ of}
\]
\[
(\text{Just } e_1, \text{Just } e_2) \to
\]
\[
d_1 \circ \text{translate } ((e_1 v + e_2 (-v)) \ast v) d_2
\]
\[
\quad \to
\]
\[
- d_1 \circ d_2
\]

**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 \( \text{beside} \) 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:

\[
\text{newtype Diagram} = \text{Diagram } (\text{Dual } [\text{Prim}], \text{Envelope})
\]
\[
\text{deriving } (\text{Semigroup, Monoid})
\]
\[
\text{unD } (\text{Diagram } (\text{Dual } ps, _-)) = ps
\]
\[
\text{prim } p = \text{Diagram } (\text{Dual } [p], \text{envelopeP } p)
\]
\[
\text{mkD } = \text{hom } \text{prim}
\]
\[
\text{envelope } (\text{Diagram } (_-, e)) = e
\]

Now combining two diagrams with \( \ast \) will result in their primitives as well as their cached envelopes being combined. However, it’s not \textit{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

\[
e = \text{hom } \text{envelopeP } ps.
\]

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

\[
\text{hom } \text{envelopeP } [\ ] = e.
\]

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

\[
e_1 \ast e_2 = \text{hom } \text{envelopeP } (ps_2 +\mathcal{C} ps_1).
\]
Again, since \( \text{hom envelope} \) is a monoid homomorphism,

\[
\text{hom envelope} (\text{ps}_2 \oplus \text{ps}_1) = \text{hom envelope} \text{ps}_2 \circ \text{hom envelope} \text{ps}_1,
\]

which by assumption is equal to \( e_2 \circ e_1 \).

But wait a minute, we wanted \( e_1 \circ e_2 \)! Never fear: \( \text{Envelope} \) actually forms a commutative monoid, which can be seen by noting that \( \text{Max} \mathbb{R} \) is a commutative semigroup, and \((\oplus) \, V_2 \) and \( \text{Option} \) both preserve commutativity.

Intuitively, it is precisely the fact that the old version of \( \text{envelope} \) (defined in terms of \( \text{hom envelope} \)) was a monoid homomorphism which allows caching \( \text{Envelope} \) 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 \( \text{beside} \). Querying the envelope of a diagram with \( n \) primitives still requires evaluating \( O(n) \) applications of \( \text{min} \), 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 \( \text{beside} \), 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.

\[
\text{newtype Trace} = \text{Trace} (P_2 \to V_2 \to \mathbb{R})
\]

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

\[
\text{newtype Trace} = \text{Trace} (P_2 \to V_2 \to \text{Min} \mathbb{R})
\]

\( \text{deriving (Semigroup)} \)

Just as with Envelope, this is a semigroup but not a monoid, since there is no largest element of \( \mathbb{R} \). Again, inserting \( \text{Option} \) will make 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 \( V_2 \)):

\[
\begin{align*}
P_2 & \to V_2 \to \text{Option} (\text{Min} \mathbb{R}) \\
P_2 & \to \text{Option} (V_2 \to \text{Min} \mathbb{R}) \\
\text{Option} (P_2 & \to V_2 \to \text{Min} \mathbb{R})
\end{align*}
\]

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:

\[
\text{newtype Trace} = \text{Trace} (P_2 \to V_2 \to \text{Option} (\text{Min} \mathbb{R}))
\]

\( \text{deriving (Semigroup, Monoid)} \)

Assuming there is a function \( \text{trace} P :: \text{Prim} \to \text{Trace} \) to compute the trace of any primitive, we could define

\[
\begin{align*}
\text{trace} & :: \text{Diagram} \to \text{Trace} \\
\text{trace} &= \text{hom trace} P \circ \text{unD}
\end{align*}
\]

However, this is a monoid homomorphism since \( \text{Trace} \) is also a commutative monoid, so we can cache the trace of each diagram as well.

\[
\begin{align*}
\text{newtype Diagram} &= \text{Diagram (Dual [Prim], Envelope, Trace)} \\
\text{deriving (Semigroup, Monoid)}
\end{align*}
\]
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:

```haskell
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: \( t_1 \circ t_2 \) represents the transformation which performs first \( t_2 \) and then \( t_1 \), and \( \varepsilon \) is the identity transformation. Given these intuitive semantics, we expect

\[
\text{transformP}\, \varepsilon\, p = p \tag{1}
\]

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

\[
\text{transformP}\, (t_1 \circ t_2)\, p = \text{transformP}\, t_1\, (\text{transformP}\, t_2\, p) \tag{2}
\]

that is, \( t_1 \circ t_2 \) 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, \( \eta \)-reducing (1) and (2) yields

\[
\text{transformP}\, \varepsilon = \text{id} \tag{1'}
\]

\[
\text{transformP}\, (t_1 \circ t_2) = \text{transformP}\, t_1 \circ \text{transformP}\, t_2 \tag{2'}
\]

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:

```haskell
class Monoid m ⇒ Action m a where
    act :: m → a → a

instance Action Transformation Prim where
    act = transformP
```

(Not 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 \( m_1, m_2 : m, \)

\[
act\, \varepsilon = id \tag{MA1}
\]

\[
act\, (m_1 \circ m_2) = act\, m_1 \circ act\, m_2 \tag{MA2}
\]

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.

```haskell
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 ○ g) = map f ○ map g.

Proof.

\[
\begin{align*}
\text{transformD}\, \varepsilon &= \{ \text{definition of transformD} \} \varepsilon \\
\text{map}\, (act\, \varepsilon) &= \{ \text{Transformation / Prim} \} \varepsilon \\
\text{map id} &= \{ \text{list functor} \} \text{id} \\
\text{transformD}\, (t_1 \circ t_2) &= \{ \text{definition} \} \text{transformation} \\
\text{map (act}\, t_1 \circ \text{act}\, t_2) &= \{ \text{list functor} \} \text{transformation} \\
\text{map (act}\, t_1 \circ \text{map (act}\, t_2) &= \{ \text{definition} \} \text{transformation} \hspace{1cm} \square
\end{align*}
\]

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

```haskell
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

\[
act\, t\, (d_1 \circ d_2) = (act\, t\, d_1) \circ (act\, t\, d_2), \tag{4}
\]

since

\[
act\, t\, (d_1 \circ d_2) = \{ \text{definitions of act and (○)} \} \\
\text{map (act}\, t)\, (d_1 \uplus d_2) = \{ \text{naturality of (++)} \} \\
\text{map (act}\, t)\, d_1 \uplus \text{map (act}\, t)\, d_2 = \{ \text{definition} \} \\
act\, t\, d_1 \circ act\, t\, d_2
\]

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:

\[
\begin{align*}
act \ m \ \varepsilon &= \varepsilon & \text{(MA3)} \\
act \ m \ (n_1 \odot n_2) &= \act \ m \ n_1 \odot \act \ m \ n_2 & \text{(MA4)}
\end{align*}
\]

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

\[
\begin{align*}
\text{act } \epsilon &= \epsilon \\
\text{act } \epsilon \odot \epsilon &= \epsilon
\end{align*}
\]

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.)

\[
\begin{align*}
\text{instance } \text{Action Transformation Envelope where} & \quad \ldots \\
\text{instance } \text{Action Transformation Trace where} & \quad \ldots
\end{align*}
\]

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

\[
e = \text{hom envelopeP } ps.
\]

We must therefore prove that

\[
\text{act } \epsilon \odot \text{hom envelopeP } ps = \text{hom envelopeP } (\text{map } (\text{act }) \ ps),
\]

or, in point-free form,

\[
\text{act } \epsilon \odot \text{hom envelopeP } = \text{hom envelopeP } \odot \text{map } (\text{act })
\]

**Proof.** We reason as follows:

\[
\begin{align*}
\text{act } \epsilon & \odot \text{hom envelopeP } P \quad \{ \text{definition} \} \\
\text{act } \epsilon \odot \text{mconcat } \odot \text{map envelopeP } P & \quad \{ \text{lemma proved below} \} \\
\text{mconcat } \odot \text{map } (\text{act }) \odot \text{map envelopeP } P & \quad \{ \text{list functor, (TE)} \} \\
\text{mconcat } \odot \text{map envelopeP } \odot \text{map } (\text{act }) & \quad \{ \text{definition} \} \\
\text{hom envelopeP } \odot \text{map } (\text{act }) & \quad \Box
\end{align*}
\]

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

**Proof.** The proof is by induction on an arbitrary list (call it \(l\)) given as an argument to \(\text{act } \epsilon \odot \text{mconcat}\). If \(l\) is the empty list,

\[
\begin{align*}
\text{act } \epsilon & \odot \text{mconcat } [] \\
& = \{ \text{mconcat } [] \} \\
& = \{ \text{monoid action (MA3)} \} \\
& = \{ \text{mconcat, definition of map } \} \\
& = \{ \text{definition of map } \} \\
& = \text{mconcat } (\text{map } (\text{act } l []))
\end{align*}
\]

In the case that \(l = x:xs\),

\[
\begin{align*}
\text{act } \epsilon & \odot \text{mconcat } (x:xs) \\
& = \{ \text{mconcat } (x:xs) \} \\
& = \{ \text{monoid action (MA4)} \} \\
& = \{ \text{induction hypothesis} \} \\
& = \{ \text{mconcat } \} \\
& = \{ \text{mconcat } \} \\
& = \text{mconcat } (\text{act } x \odot \text{mconcat } (\text{act } t ) xs) \\
& = \{ \text{definition of map } \} \\
& = \text{mconcat } (\text{map } (\text{act } t ) (x:xs))
\end{align*}
\]

**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 \(\text{beside}\) repeatedly to lay out a line of diagrams, as in the following code (whose result is shown in Figure 10):

\[
\begin{align*}
\text{unit}_2 &: \mathbb{V}_2 & \text{— unit vector along the positive x-axis} \\
\text{hcat} & = \text{foldr} (\text{beside } \text{unit}_2) \varepsilon \\
\text{lineOfCircles } n & = \text{hcat } (\text{replicate } n \text{ circle})
\end{align*}
\]

Fully evaluating \(\text{lineOfCircles } n\) takes \(O(n^2)\) time, because the \(\text{foldr}\) call to \(\text{beside}\) must map over \(n\) primitives, resulting in \(1 + 2 + 3 + \cdots + n\) total calls to \(\text{transformP}\). (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

\[
\text{act } (t_1 \odot t_2) = \text{act } t_1 \odot \text{act } t_2.
\]

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:

```haskell
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:

```haskell
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:

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

So, transformations are delayed somewhat—but only until a call to (∙), 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 refi ned 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:

```haskell
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 (∙):

```haskell
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):

```haskell
envelope :: Diagram → Envelope
envelope (Prim p) = envelopeP p
envelope Empty = ε
envelope (Compose (ε,ε)) = ε
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:

```haskell
instance Semigroup Diagram where
  Empty ⊙ d = d
  d1 ⊙ d2 = Compose (envelope d1 ⊙ envelope d2,
                      trace d1 ⊙ trace d2)
  d₁ d₂
```

```haskell
instance Monoid Diagram where
  ε = Empty
```

The particularly attentive reader may have noticed something strange about this Semigroup instance: (∙) is not associative! d₁ ⊙ (d₂ ⊙ d₃) and (d₁ ⊙ d₂) ⊙ d₃ are not equal, since they result in trees of two different shapes. However, intuitively it seems that d₁ ⊙ (d₂ ⊙ d₃) and (d₁ ⊙ d₂) ⊙ d₃ 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 d₁ and d₂ equivalent whenever \( unD d₁ \equiv unD d₂ \), where \( unD :: \text{Diagram} \rightarrow [\text{Prim}] \)

```haskell
“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:

```haskell
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:

```haskell
unD :: Diagram → [Prim]
unD (Prim p) = [p]
unD Empty = ε
unD (Compose d₁ d₂) = unD d₂ \circ 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 ε (unD d) = go ε d = unD' d.

Proof. By induction on d.

- If d = Prim p, then act t (unD (Prim p)) = act t [p] = [act t p] =
  go t (Prim p).

- If d = Empty, then act t (unD Empty) = act t ε = ε = go t Empty,
  where the central equality is (MA3).

- If d = Compose c d1 d2, then
  
  act t (unD (Compose c d1 d2))
  = \{ definition \}
  act t (unD d2 ∘ unD d1)
  = \{ monoid action (MA4) \}
  act t (unD d2) ∘ act t (unD d1)
  = \{ induction hypothesis \}
  go t d2 ∘ go t d1
  = \{ definition \}
  go t (Compose c d1 d2)

- Finally, if d = Act t' d', then
  
  act t (unD (Act t' d'))
  = \{ definition \}
  act t (act t' (unD d'))
  = \{ monoid action (MA2) \}
  act (t ∘ t') (unD d')
  = \{ induction hypothesis \}
  go (t ∘ t') d'
  = \{ definition \}
  go t (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

\( d_1 \circ (d_2 \circ (d_3 \circ d_4)) \)

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 \((xss++)::[a] \to [a]\). 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:

\[
((xss++) \circ (ys++) \circ (zs++)) [] = xss++ (ys++ (zs++ [])).
\]

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 \((++) \circ ε = ε\), which expands to \((++) [] = id\), that is, \([a] + xs = xs\), which is true by definition. (H2) states that \((++) (xs \circ ys) = (+++) xs \circ (++) ys\), 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 \(unD\):

\[
\text{unD''} :: \text{Diagram} \to [\text{Prim}]
\]

\[
\text{unD''} d = \text{appEndo} (\text{go ε d}) [\] \text{where}
\]

\[
go :: \text{Transformation} \to \text{Diagram} \to \text{Endo} [\text{Prim}]
\]

\[
go t (\text{Prim p}) = \text{Endo} ([\text{act t p}])
\]

\[
go _\text{Empty} = \varepsilon
\]

\[
go t (\text{Compose } _ c d1 d2) = go t d2 \circ go t d1
\]

\[
go t (\text{Act } t' d) = go (t \circ t') d
\]

**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”:

\[
data \ MTree \ u \ l = \text{Leaf} u \ l
\]

\[
| \text{Empty}
\]

\[
| \text{Compose } u \ (\text{MTree } d \ u \ l) \ (\text{MTree } d \ u \ l)
\]

\[
| \text{Act } d \ (\text{MTree } d \ u \ l)
\]

d represents a “downwards-travelling” monoid, which acts on the structure and accumulates along paths from the root. \(u\) represents an “upwards-travelling” 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:

\[
type \ \text{Diagram} = \text{MTree} \text{Transformation} \ (\text{Envelope}, \text{Trace}) \ \text{Prim}
\]

\[
\text{prim p = Leaf} (\text{envelopeP p}, \text{traceP p}) p
\]

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:

\[
\text{instance } (\text{Action } m a, \text{Action } m b)
\Rightarrow \text{Action } (m,a) \ (b) \ where
\]

\[
\text{act } m \ (a,b) = (\text{act } m \ a, \text{act } m \ b)
\]

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 \( l \rightarrow u \) 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:

\[
\begin{align*}
gut \colon &: (\text{Action} \, d \, u, \text{Monoid} \, u) \Rightarrow \text{MTree} \, d \, u \, \mapsto \, u \\
gut \, (\text{Leaf} \, u) &= \, u \\
gut \, \text{Empty} &= \, e \\
gut \, (\text{Compose} \, u \, \_ \, = \, u \\
gut \, \text{Act} \, d \, t &= \, \text{act} \, d \, (\text{gut} \, t) \\
envelope &= \, \text{fst} \, \circ \, \text{gut} \\
trace &= \, \text{snd} \, \circ \, \text{gut}
\end{align*}
\]

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

\[
\text{instance \, (Action \, d \, u, \text{Monoid} \, u) \Rightarrow \text{Semigroup} \, (\text{MTree} \, d \, u \, l) \, \text{where}}
\]
\[
\begin{align*}
\text{Empty} \circ l &= t \\
(\text{Empty}) \circ t &= t \\
(\text{t1}) \circ (\text{t2}) &= \text{Compose} \, (\text{gut} \, \text{t1} \, \circ \, \text{gut} \, \text{t2}) \\
\text{act} \_ \, \text{Empty} &= \text{Empty} \\
\text{act} \, d \, \text{Act} \, d' \, t &= \text{act} \, \text{d} \, (\text{d} \, \circ \, d') \, t \\
\text{act} \, d \, t &= \text{Act} \, d \, t
\end{align*}
\]

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”: \((\circ)\) itself is a monoid homomorphism.

\[
\begin{align*}
\text{foldMTree} \colon &: (\text{Monoid} \, d, \text{Monoid} \, r, \text{Action} \, d \, r) \Rightarrow (l \rightarrow r) \rightarrow \text{MTree} \, d \, u \, l \rightarrow r \\
\text{foldMTree} \, \text{leaf} \, t &= \text{appEndo} \, (\text{go} \, e \, t) \, e \text{ where} \\
\text{go} \, d \, (\text{Leaf} \, l) &= \text{Endo} \, (\text{act} \, d \, (\text{leaf} \, l) \circ) \\
\text{go} \_ \, \text{Empty} &= e \\
\text{go} \, d \, (\text{Compose} \, t \, t') &= \text{go} \, d \, t \, \circ \, \text{go} \, d \, t' \\
\text{go} \, d \, (\text{Act} \, d' \, t) &= \text{go} \, (\text{d} \, \circ \, d') \, t \\
\text{unD} \colon &: \text{Diagram} \rightarrow [\text{Prim}] \\
\text{unD} &= \text{getDual} \, \circ \, \text{foldMTree} \, (\text{Dual} \, (\circ))
\end{align*}
\]

Again, associativity of \((\circ)\) 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 \(e\) corresponding to the Style containing no attributes, and \((\circ)\) corresponding to right-biased union. For example,

\[
\begin{align*}
\{\text{Fill Purple, Stroke Red}\} \circ \{\text{Stroke Green, Alpha } 0.3\} &= \{\text{Fill Purple, Stroke Green, Alpha } 0.3\}
\end{align*}
\]

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

\[
\text{applyStyle} \colon \text{Style} \rightarrow \text{Diagram} \rightarrow \text{Diagram}
\]

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:

\[
\text{type Diagram} = \text{MTree} \, (\text{Transformation}, \text{Style}) \, (\text{Envelope}, \text{Trace}) \, \text{Prim}
\]

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:

\[
\text{instance Action Style Envelope where}
\]
\[
\text{act} \_ = \text{id}
\]

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

\[
\text{instance \, (Action} \, m_1 \, a, \text{Action} \, m_2 \, a) \Rightarrow \text{Action} \, (m_1, m_2) \, a \text{ where}
\]
\[
\text{act} \, (m_1, m_2) = \text{act} \, m_1 \, \circ \, \text{act} \, m_2
\]

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

\[
\text{act} \, ((m_1, m_2) \circ (m_1, m_2)) = \text{act} \, (m_1, m_2) \, \circ \, \text{act} \, (m_1, m_2).
\]

Beginning with the left-hand side,

\[
\begin{align*}
\text{act} \, ((m_1, m_2) \, \circ \, (m_1, m_2)) &= \{\text{product monoid}\} \\
\text{act} \, (m_1, m_2) \, \circ \, (m_1, m_2) &= \{\text{proposed definition of act for pairs}\} \\
\text{act} \, (m_1, m_2) \, \circ \, (m_1, m_2) &= \{\text{m_1, m_2, m_1, m_2 (MA2)}\} \\
\text{act} \, m_1 \, \circ \, m_2 \, \circ \, m_1 \, \circ \, m_2 &= \text{act} \, m_1 \, \circ \, \text{act} \, m_2 \, \circ \, \text{act} \, m_2 \\
\end{align*}
\]

But the right-hand side yields

\[
\begin{align*}
\text{act} \, ((m_1, m_2) \, \circ \, (m_1, m_2)) &= \{\text{proposed definition of act}\} \\
\text{act} \, m_1 \, \circ \, m_2 \, \circ \, m_1 \, \circ \, m_2 &= \text{act} \, m_1 \, \circ \, \text{act} \, m_2 \, \circ \, \text{act} \, m_2
\end{align*}
\]

In general, these will be equal only when \(\text{act} \, m_1 \, \circ \, \text{act} \, m_2 = \text{act} \, m_2 \, \circ \, \text{act} \, m_1 \)—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

\[
\text{type Diagram} = \text{MTree} \, (\text{Transformation}, \text{Style}) \, (\text{Envelope}, \text{Trace}) \, \text{Prim}
\]

**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 the 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 Sjoberg for being willing to listen to my ramblings about diagrams and for offering 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 an early version 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 \( \text{mconcat} = \text{foldr} (\cdot) \varepsilon \), we compute \( \text{mconcat} [] = \text{foldr} (\cdot) \varepsilon [] = \varepsilon \), and

\[
\text{mconcat} (x:xs) = = \text{foldr} (\cdot) \varepsilon (x:xs) = x \cdot \text{foldr} (\cdot) \varepsilon xs = x \cdot \text{mconcat} xs.
\]

These facts are referenced in proof justification steps by the hint \( \text{mconcat} \).

First, recall the definition of \( \text{hom} \), namely

\[
\text{hom} :: \text{Monoid} m \Rightarrow (a \rightarrow m) \rightarrow ([a] \rightarrow m)
\]

\[
\text{hom} f = \text{mconcat} \circ \text{map} f.
\]

We first note that

\[
\text{hom} f (x:xs)
\]

\[
= \{ \text{ definition of hom and map } \}
\]

\[
\text{mconcat} (f x: \text{map} f xs)
\]

\[
= \{ \text{ mconcat }\}
\]

\[
f x \cdot \text{mconcat} (\text{map} f xs)
\]

\[
= \{ \text{ definition of hom } \}
\]

\[
f x \cdot \text{hom} f xs
\]

We now prove that \( \text{hom} f \) is a monoid homomorphism for all \( f \).

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

Second, we show (H2), namely,

\[
\text{hom} f (x:ys) = \text{hom} f xs \cdot \text{hom} f ys,
\]

by induction on \( xs \).

- If \( xs = [] \), we have \( \text{hom} f ([] ++ ys) = \text{hom} f ys = \varepsilon \cdot \text{hom} f ys = \text{hom} f ys \).
- Next, suppose \( xs = x:xs' \):

\[
\text{hom} f ((x:xs') ++ ys)
\]

\[
= \{ \text{ definition of } (++) \}
\]

\[
= \text{hom} f (x:(xs' ++ ys))
\]

\[
= \{ \text{ hom of } (:) \}, \text{ proved above } \}
\]

\[
= f \cdot \text{hom} f (xs' ++ ys)
\]

\[
= \{ \text{ induction hypothesis } \}
\]

\[
= f \cdot \text{hom} f xs' \cdot \text{hom} f ys
\]

\[
= \{ \text{ associativity of } (\cdot) \text{ and hom of } (:) \}
\]

\[
= (\text{hom} f (x:xs')) \cdot \text{hom} f ys
\]

As a corollary, \( \text{mconcat} (xs ++ ys) = \text{mconcat} xs ++ \text{mconcat} ys \), since \( \text{hom} id = \text{mconcat} \circ \text{map} id = \text{mconcat} \).