Hulk Smash

Emily Pillmore

August 1, 2020

Introduction

Some Geometry

Translation to Haskell

What about These?

Tying it all together

Introduction

My name is Emily Pillmore.

I am a Haskell and Math enthusiast. Trying to be more of the latter, less of the former lately.

- Twitter (@pitopos)
- Meetups in NYC: NY Homotopy Type Theory, NY Category Theory, and the NY Haskell User Group.
- Meetups in Asheville: Asheville Functional Programming Groups, Asheville Category Theory (only member :'()
- Discord: Haskell $\cap$ Dank Memes: https://discord.gg/2x2fYSK.
- Personal: All of my slides and meetup content are hosted at cohomolo.gy.

If you ever want to talk category theory or programming, I'm around to talk, help, etc.

I got my start in Scala, with Runar's *Functional Programming in Scala* (the red one - there is now a blue one)

I work in Haskell pretty exclusively now, recreationally with Coq and Lean.

I work at a company called **Kadena**, designing a language, a private permissioned blockchain, and a public one.

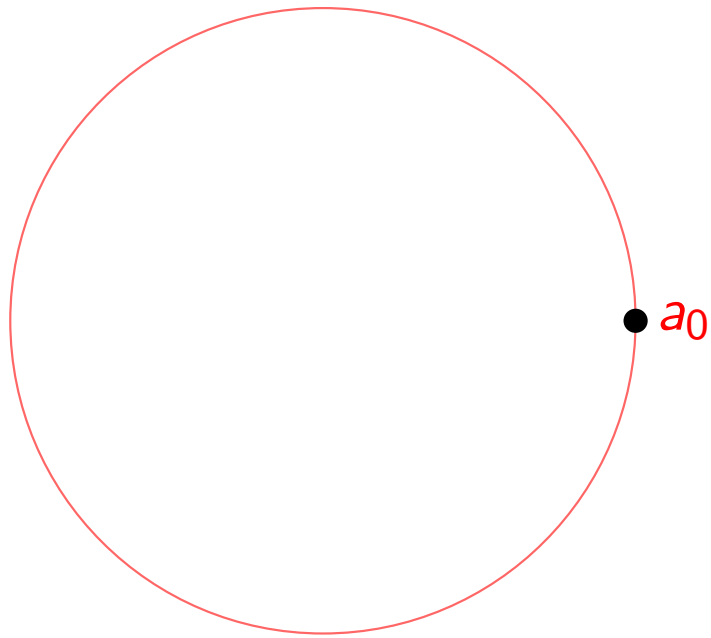
- A lite talk to end the day - nothing too serious or strenuous
- To learn a bit more about the Maybe monad than you probably knew prior to this talk
- See an interesting geometric perspective on the subject
- Take away some inspiration for thinking about types in Haskell
- Have fun and spread the... like.

To enjoy ourselves

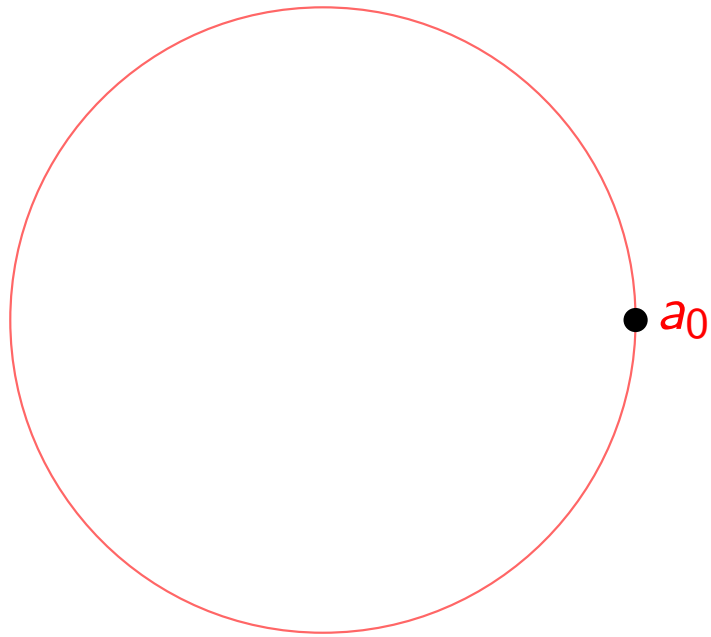
Some Geometry

In classical point-set topology we often consider topology and geometry in terms of sets of points.

Often, we consider geometric objects with a particular "distinguished" base point picked out for us.



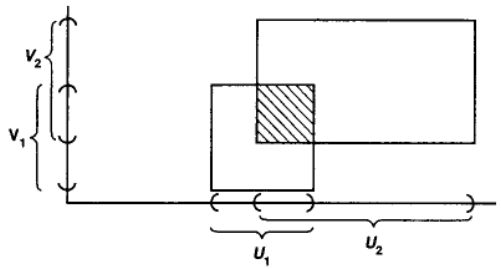
Often, we consider geometric objects with a particular "distinguished" base point picked out for us.

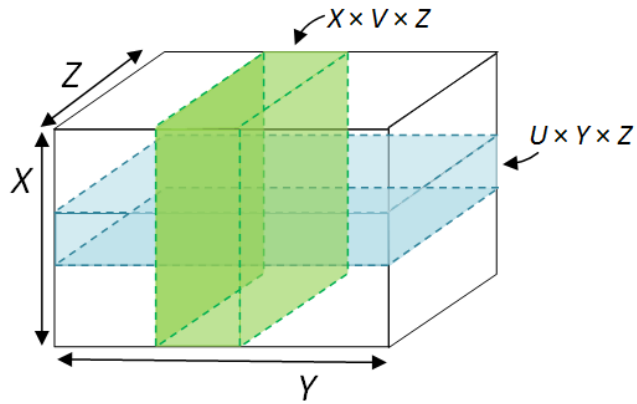


we can think of this as a pair of a geometric object and a base point (A, a_0).

We can build our usual product, coproduct and quotient spaces, and we can use the action of the base point to orient ourselves within the space.

Products are relatively simple. We want to build a cartesian product of spaces as you already know how to do.

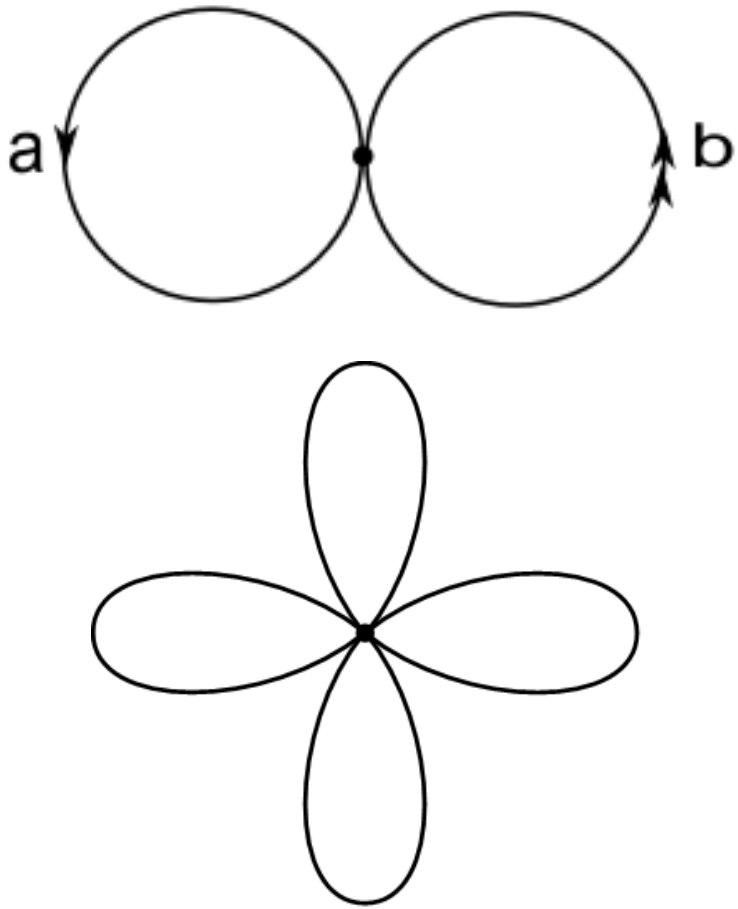




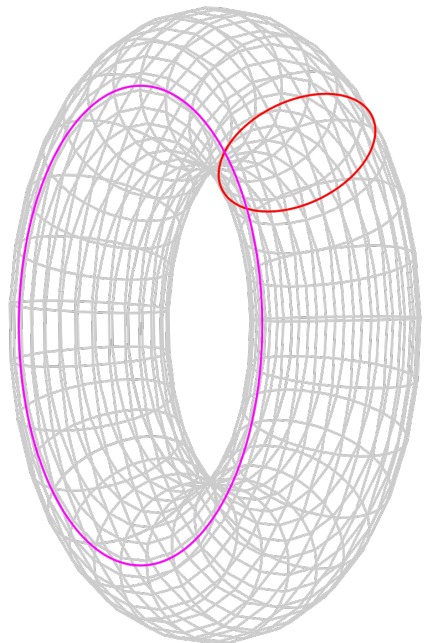
What would a point look like in this product space? A pointed product? coproduct? A pointed coproduct?

Quotient spaces are spaces built by identifying points according to some relation. Get ready for some bad paper-taping action

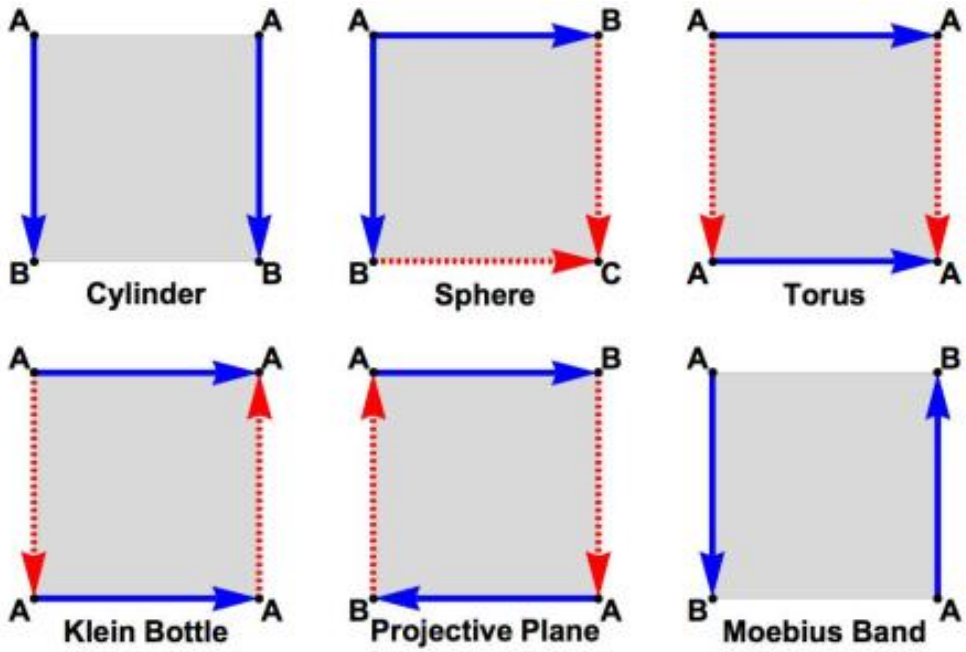
Some quotients can be very simple. For instance, the wedge sum (pointed coproduct) of two spaces is built by identifying the distinct basepoints of each object. We use the notation $X \vee Y$. When the objects are 1-spheres, we call this a *topological rose*.



Other quotients can be more complex. For example, consider the square torus
$\mathbb{T} \cong [0,1] \times [0,1]/(x,0) \sim (x,1), (0,y) \sim (1,y)$:



If you want to try it yourself, it's actually quite easy with some paper and tape:



I'd be remiss not to mention the *smash product*, or *collapsed product* - the quotient space of a product by a wedge sum. Formally, the smash product X and Y is the quotient $X \wedge Y := X \times Y / X \vee Y$

Translation to Haskell

In Haskell, we have discrete analogues of these constructions available to us, but first, we must consider the monad in which we will work: the **Maybe** monad.

```haskell
data Maybe a = Just a | Nothing

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)

instance Applicative Maybe where
  pure = Just

  Just f <*> a = fmap f a
  Nothing <*> _ = Nothing

instance Monad Maybe where
  return = Pure

  Just a >>= f = f a
  Nothing >>= _ = Nothing
```

The **Maybe** monad corresponds with the map $A \mapsto A \sqcup *$, which can be thought of as "adjoining a basepoint" to A. We can consider **MaybeA** to be a "pointed object" in the precise that it is an object of **Hask** equipped with a disjoint basepoint.

Using this fact, we can draw a direct line from point-set topology and the objects we just discussed to Haskell types, using a general notion of "pointed", and looking to category theory and type arithmetic.

Products of pointed spaces have a straightforward calculus (technically, abusing RAPL/LAPC/Yoneda all over the place):

```
(Maybe a, Maybe b)
-- simplifying notation
~ (1 + a) * (1 + b)
-- products distribute over coproducts
~ 1 + b + a + a*b
-- associativity
~ 1 + (a + b + a*b)
~ Maybe (Either a (Either b (a,b)))
```

Those in the know will recognize the term $a + b + a * b$ as the **These** datatype.

```haskell
data These a b
  = This a
  | That b
  | These a b
```

More on **These** later. But, to recap, a "product of pointed objects" is isomorphic to **Maybe (These a b)**.

Pointed coproducts are slightly easier. A pointed coproduct is a coproduct of pointed objects, but with the base point identified. Intuition: attaching a basepoint to the usual coproduct diagrams.

```
Either (Maybe a) (Maybe b)
-- simplification of notation
~ (1 + a) + (1 + b)
-- basepoints are identified
-- via the pushout: a0 <- * -> b0
~ 1 + a + b
-- assocativity
~ 1 + (a + b)
~ Maybe (Either a b)
```

This is precisely the shape of a wedge sum, like the topological rose we just saw!

We can even form an analogue of the smash product we build. Recall that the *smash product* of X and Y is the quotient of the product by a wedge: $X \wedge Y := X \times Y / X \vee Y$.

```
(Maybe a, Maybe b) / Maybe (Either a b)
-- notation
~ (1 + a) * (1 + b) / (1 + a + b)
-- defn + reassociate
~ (1 + a + b) + a*b / (1 + a + b)
-- quotient (1 + a + b) ~ * - note that
-- this means 'identify to a point', not divide!
~ 1 + a*b
~ Maybe (a,b)
```

This means to build the geometry above, we need the following three datatypes (forgive the names)

```
data Can a b = Non | One a | Eno b | Two a b
data Wedge a b = Nada | Here a | There b
data Smash a b = Nothing | Smash a b
```

So this is pretty interesting! We're not actually skimping on details to build this calculus, just some good knowledge of how pointed objects work, and geometry to guide our intuition.

CT Aside: When the ambient category is closed monoidal, smash products form a separate monoidal tensor in its category of pointed objects, with its own base point-preserving analogue to currying, uncurrying (see: basepoint-preserving function spaces), and the usual distributivity over pointed coproducts (Wedges) and symmetry + associativity that one might expect. When the ambient category is additionally symmetric , the smash product is also symmetric.

What about These?

What's up with **These**? It doesn't really seem to fit in:

- It's got no "basepoint" to speak of
- There's no real good intuition for it, unlike the rest other than maybe "inclusive union"
- It was originally developed to encapsulate "zippy" behavior.
- How can we think of it beyond being a "convenient data type that fits with our ad hoc needs"?
- Is **These** to the pointed product as **Semigroup** is to **Monoid**?

Fact: **These** is not so ad hoc as it might seem. In fact, there's some principles hiding behind the scenes.

The encoding we've been implicitly using to denote "pointed objects" has been:

- objects are pairs (A, a_0)
- morphisms $f : A \to B$ such that $fa_0 = b_0$.

However, there is an alternate encoding that is equivalent.

Consider this equivalent encoding of pointed spaces:

- objects are the same
- morphisms $f : A \to B \sqcup *$

These encodings are, in a loose sense, equivalent by the following functor **F**:

- For objects: $FA = (A \sqcup *, *)$
- For morphisms $f : A \to B \sqcup * = (>>= f)$
- and any morphism $g : A \sqcup * \to B \sqcup *$ which satisfies $g* = *$ corresponds with f such that $g = (>>= f)$

Note: this is "loose" in the sense that types are not sets and one must treat them like they are to make that equivalence work. In the first encoding, $(Int \to Int, id)$ is an object, but there's no type A such that $A \sqcup * \; Int \to Int$.

So what is **These**? Another way of encoding the pointed product, using the Kleisli category of the ‘Maybe‘ monad.

Which is nicer? You decide.

Tying it all together

Geometry gives us great intuition to draw from.

We can look to Category theory not just to generalize and abstract our results, but translate across mathematical fields and boundaries.

Take ideas from everywhere, filter through category theory and sort into whatever hat you want to wear.

All of these constructs are encapsulated in the smash suite of libraries.

Feel free to contribute. There's lots of work to be done in terms of implementing monad transformers, optics, fun combinators
etc.

Thanks to Mniip for making the offhand comment that started me on this kick. And thanks to Haskell Love for tolerating me!