The Algebra of ADT's

SIG Math Fall 2025

The only stuff that's really relevant is Section I. The rest is insane math that is fun to think about but probably not useful.

Section I.
What is a type?

- There are a number of ways to define what a type is, including things like what information it has, its memory layout, etc.
- Today we are focused on the mathematical side of things, so our definition of a type is the **set of values a variable could take on.**

Types explained

- Types provide assurance to the programmer that a variable will only be in a certain range of values.
- If you have two variables that are both integers, you know for a fact adding them won't cause an error.
    - For instance, a+b works fine if a = 5 and b = 6, but if a = 5 and b = “foo” there will be an issue!
- Generally if your code has a type mismatch, it will not run (e.g. const blah: number = "test"). This helps cut down on “obvious” bugs.
- Types were initially invented as a way to reduce bugs, but theoretical computer scientists reformulated much of computer science foundations in types (known as type theory)

Examples of types

- The easiest way to construct a type is to just list the values it can take on:
  - Boolean (true, false)
  - Unsigned 8-bit integer (0 through 255)
  - Integer (0, 1, -1, 2, -2, 10000000, etc)
  - Floating-point number (all possible combinations of 64 bits)
- In many languages, integers and floating point numbers have different types based on how many bits they have and whether they are signed. Because this is a SIG Math lecture, we will just refer to all integers as Ints that could theoretically be unbounded.

Types in programming languages

- Some modern languages let you construct your own types by enumerating the values it can hold (e.g. TypeScript, Rust, Haskell)

```
data AwesomeNumbers = 21 | 67 | 521 | 893;
data PoliticalParty = Democrat | Republican | Independent;
```

- We can also **construct types that include other types**:

```
data BoolOrIDK = IDontKnow | Boolean;
```
(equivalent to `IDontKnow | True | False`)

- Notice that **we can create new values out of nothing**. `IDontKnow` is just its own value, it doesn't equal anything else.
- The pipe key **`|` means or** (we will cover this in a future slide)

Higher kinded types

- We can even make types that take another type as an argument.
- The most common example is an array -- an array isn't a type on its own, but `int[]` and `double[]` are.

```
data Nullable a = a | Null
data Option a b = a | b
```

- These are two real types from the Rust programming language and they are very commonly used!

Cardinality

- The cardinality of a type is how many values it can represent.

```
data Bool = True | False;
data AwesomeNumbers = 21 | 67 | 521 | 893;
data Nullable a = a | Null;
data Option a b = a | b;
```

- We're only focused on finite cardinalities for the purposes of this lecture.

- True
  - Cardinality 1
- Bool
  - Cardinality 2
- AwesomeNumbers
  - Cardinality 4
- Nullable Bool
  - Cardinality 3 (2 from Bool, 1 from Null)
- Option Bool AwesomeNumbers
  - Cardinality 6 (2 + 4)

The name for constructing a type using **| (or) is a sum type**. When finding the cardinality of a sum type, you **add the cardinalities of the two types**.

Compound types

- We can also construct types that require two types together. In this example, HTTPResult is constructed by a boolean and a StatusCode:

```
type StatusCode = 200 | 400 | 500;
data HTTPResult = {
    isError :: Bool;
    status :: StatusCode;
}
```

- What is the cardinality of HTTPResult?
- We can find the cardinality of all its members, and multiply them.
- Consequently, a struct/class/record type like HTTPResult is known as a **product type**.

Recap

- To recap:
  - **Sum** types are type A **OR** type B, cardinalities are **added**
  - **Product** types are type A **AND** type B, cardinalities are **multiplied**
- Challenge: what is the cardinality of a function a -> b?
  - Answer: b^a. Try working it out for Bool -> StatusCode yourself
- If you were here for the lecture on tropical algebra, you might remember the concept of a semiring, a set where you can add and multiply elements and get another element.
  - Semirings also feature distributivity and commutativity, which you can prove for yourself if you want.

The rules we've covered so far actually are enough to make a very rich type system! But not every application is super intuitive.

Section II. Algebra on types

Lists, mathematically

- The mathematical structure for a list is very similar to a linked list data structure you learn in a CS class.

```
data List a = Nil | Cons a (List a)
[] -> Nil
[1] -> Cons 1 Nil
[1, 2] -> Cons 1 (Cons 2 Nil)
[1, 2, 3] -> Cons 1 (Cons 2 (Cons 3 Nil))
```

- Basically, each element in the list stores its value and the next value. The next value could either be a smaller list (which is the next value recursively), or nothing (so it's the last element).

Applying type theory

- We can also use type theory to get an insight into the nature of our datatypes. Consider the type:
  `data Choice a = LeftChoice a | RightChoice a`
  - Note: `LeftChoice` and `RightChoice` are just labels to identify which choice you're picking. It's still just an a either way. A `LeftChoice Bool` is equivalent to a `Bool`
- The cardinality of this type is a + a, since it's a sum type.
- However, note that basic algebra says a + a = 2 * a. This implies that Choice should be isomorphic (the same as) a product type of a type with cardinality 2 (like a Bool) and a:

```
type Choice a = (Bool, a)
```

Applying type theory example

- Look at the two type definitions and see if you can convince yourself that they are basically the same.

```
type HTTPResult1 a = Success a | Error a;
type HTTPResult2 = {
    success :: Bool;
    value :: a;
}
```

Everything before this is genuinely useful, and I think it's a shame that NJIT doesn't talk much about type systems or functional programming at all.

What comes next is the insane part that you should probably take with a grain of salt.

Lists revisited

- Let's look again at our list datatype:
  `data List a = Nil | Cons a (List a)`
- Let's say the cardinality of List is L, and the cardinality of a is A. Based on our knowledge of sum and product types, we can write this formula:
  `L = 1 + (a * L)`
- Note that L is on both sides. We can work with this in two ways...

The somewhat sane way

- The first way is to just substitute L back in for itself:
  $L = 1 + (A * L)$
  $L = 1 + (A * (1 + (A * L)) = 1 + a + (A^2 * L)$
  $L = 1 + A + (A^2 * (1 + (A * L)) = 1 + A + A^2 + A^3 * L$
  ...
  $L = 1 + A + A^2 + A^3 + A^4 + ...$
- This actually makes a lot of sense! The list could be empty, it could have 1 element, it could have 2 elements, it could have 3, etc.
- Each term, when evaluated with a value of A, corresponds to the number of options -- there are 2 possible 1-element lists of booleans, 4 possible 2-element lists of booleans, etc.

The insane way

- But we can also manipulate the equation in a more algebraic fashion:
  $L = 1 + (A * L)$
  $L - (A * L) = 1$
  $L * (1 - A) = 1$
  $L = 1 / (1 - A)$
- Here is the kicker: the Taylor Series for $1 / (1 - A)$ is $1 + A + A^2 + \ldots$, so we can use that expansion:
  $L = 1 + A + A^2 + \ldots$
- The result is correct, but the steps are weird: what does it mean to have a negative or fractional type? Surprisingly, this is actually rigorous!

Binary trees revisited

- Here is a data structure for a binary tree. We will apply a similar process as for lists, solving for B and using its Taylor series.
- data BinaryTree a = Leaf a | Branch (BinaryTree a) (BinaryTree a)

  $B = a + B^2$ (find cardinality)

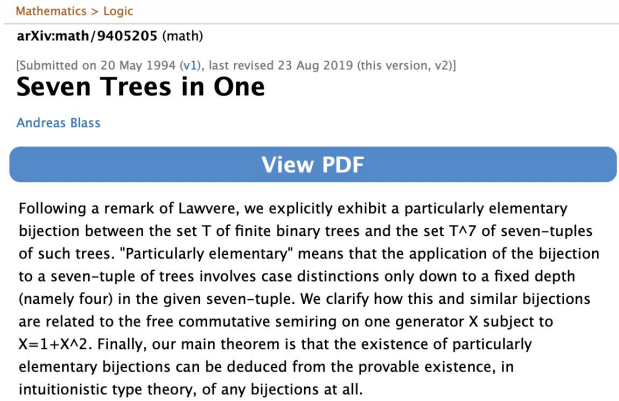
  $B^2 - B + a = 0$ (rewrite as a quadratic)

  $B = (1 - \mathrm{sqrt}(1 - 4A)) / 2$ (use quadratic formula)

  $B = A + A^2 + 2A^3 + 5A^4 + 14A^5 + \ldots$ (find Taylor series)
- The results are remarkable! By looking at the coefficients we can see that there is 1 binary tree with 2 leaves, 2 with 3 leaves, 5 with 4 leaves, and 14 with 5 leaves!

An even crazier result

- Let's take simple unmarked binary trees (where all the nodes don't have labels, so they are interchangeable). This is equivalent to the cardinality of a being 1. Then we can substitute into our equation:
  B = (1 - sqrt(1 - 4)) / 2 (note the negative sqrt)
  B = (1/2) + i * (3/2) = $e^{ipi/3}$ (use the complex exp.)
- What's notable about B here is that it satisfies the identity $B^7 = B$.
- Doesn't sound useful on its own, but it's actually a well-known fact that there is a natural bijection between binary trees and 7-tuples of binary trees!


Mathematics > Logic

arXiv:math/9405205 (math)

[Submitted on 20 May 1994 (v1), last revised 23 Aug 2019 (this version, v2)]

Seven Trees in One

Andreas Blass

View PDF

Following a remark of Lawvere, we explicitly exhibit a particularly elementary bijection between the set T of finite binary trees and the set T^7 of seven-tuples of such trees. "Particularly elementary" means that the application of the bijection to a seven-tuple of trees involves case distinctions only down to a fixed depth (namely four) in the given seven-tuple. We clarify how this and similar bijections are related to the free commutative semiring on one generator X subject to X=1+X^2. Finally, our main theorem is that the existence of particularly elementary bijections can be deduced from the provable existence, in intuitionistic type theory, of any bijections at all.

Let's prove more theorems

- Here is another example. A rose tree is similar to a binary tree, but each node can have any number of children. We can write this as:
  `data Rose a = a * List (Rose a)`
- We can use the fact that the cardinality of List a is 1 / (1 - a), from before:

```
R = a * (1 / (1 - R))          (substitute List)
R = a / (1 - R)                (simplify)
R^2 - R + a = 0                (collect terms)
```

- Notice that we got the same formula for rose trees (R^2 - R + a = 0) as we did for binary trees (B^2 - B + a = 0).
- So there must be a natural bijection between rose trees and binary trees, and there is: one of the first discoveries made in LISP was that any tree can just be encoded with Cons cells (like a list or a binary tree).

Ok so what just happened

- That was a lot of stuff and it's confusing the first time you look at it, so if you're confused I would look at the slides (or ask me to go over it again!)
- But the main idea is that from our definitions of sum type and product type, we can do all sorts of "normal math" and get some results that actually make sense.
- These results align with results that were landmark discoveries made by computer scientists in the 1950s.
- Right now we've just been doing algebra. Let's try and do calculus on types!

Section III. Calculus on types

Poking holes

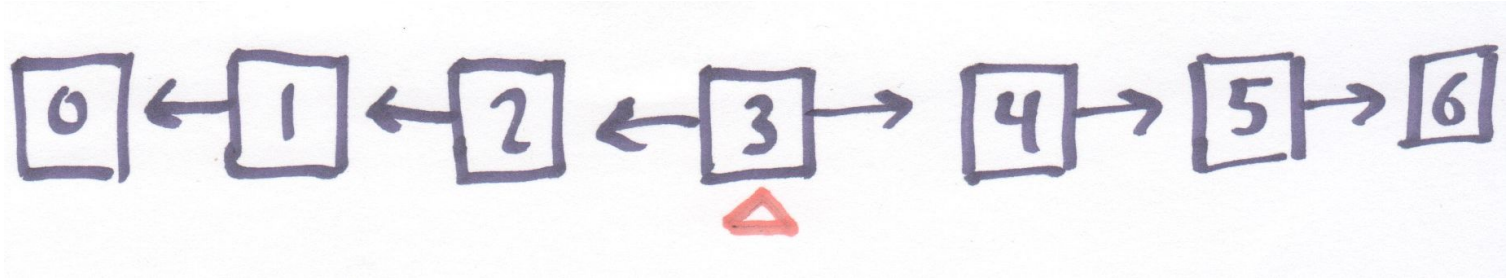
- It's almost time to do calculus on data types, but we need to introduce one last concept -- poking holes in types.
- What is a hole? Just find a spot where your type can go, and place a hole.
- How many ways can we poke an a-hole?
  a^2 -> (_, a), (a, _)
  a^3 -> (_, a, a), (a, _, a), (a, a, _)
  a*b -> (_, b) (remember, we are only poking holes of type a)
  Option a a -> Left _, Right _
  Option a b -> Left _
  b -> none!
  (Option a b) * b -> ((Left _), b)

Some observations...

- Let's make some observations:
  - Data types with no a's (constants) have no holes of type a.
  - The number of holes in a sum type is the number of holes on each side.
  - The number of holes in a product type is the number of positions in the product times the number of ways to make a hole in each of those positions.
- These explanations are kind of confusing in words, so let's write them as formulas. Let d/da be "the number of holes of type a", and let f(a) be the number of ways to poke a hole in type f.
  - d/da C = 0
  - d/da (f(a) + g(a)) = d/da f(a) + d/da g(a)
  - d/da (f(a) * g(a)) = d/da f(a) * g(a) + f(a) * d/da g(a)

Zippers

- The zipper is a way to "focus" on one element of a data structure, typically used to edit it efficiently. We can use a list to demonstrate:



- There is a focused element (3 in this case), but we can move the focused element left (to 2), right (to 4), or edit the focused element.
- This makes it O(1) for many operations instead of O(n).

Zippers mathematically

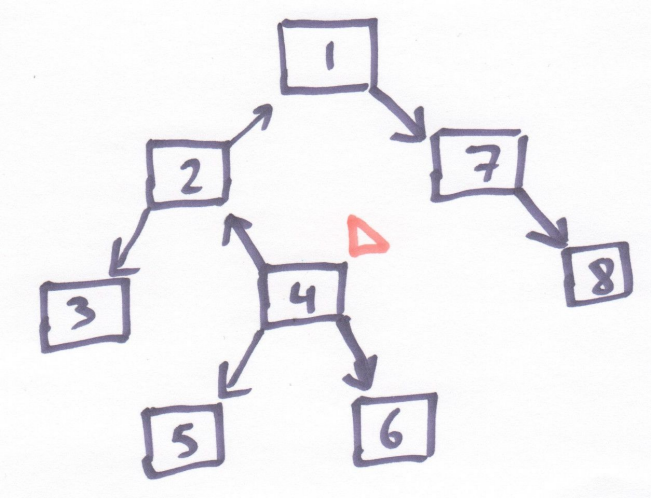
- So how can we think about zippers mathematically? Well, remember that a list is just:
  $L = 1 + aL$
- We can poke a hole by taking the derivative:
  $dL/dA = d/da(1 + aL)$ (take derivative)
  $dL/dA = L + a*dL/dA$ (product rule, 1 is a constant)
  $dL/dA = L / (1 - a)$ (factor out dL/dA)
  $dL/dA = L * 1 / (1 - a)$ (definition of division)
  $dL/dA = L^2$ (remember that $L = 1 / (1 - a)$)
- What does this mean? Well, from our diagram, L^2 is exactly the type of our zippered list -- there's one list pointing left and one pointing right.

What did we just do?

- This might be quite confusing, so I will give a few rephrasings:
- Suppose you have a list 0 through 10. Now let's poke a hole at 4. We could also say that the hole is determined by two lists (0, 1, 2, 3) and (5, 6, 7, 8, 9, 10). Both the part to the left and the part to the right are just two arbitrary lists (could be empty, if the hole is at 0 or 10).
- Of course, Euler could have told us this immediately:

$$\partial \mathrm{List}(x) = \partial(1-x)^{-1} = (1-x)^{-2} = \mathrm{List}^2(x)$$

Zippers in real life



- You actually use zippers every day in your file explorer! You can think of the file system as a rose tree (folders can contain files, or other folders).
- When you open up a particular folder, that folder is the focus of the zipper, and you can easily move to the parent or a child folder.
- The name for this type of zipper is Huet's zipper. On the right is the derivative of a binary tree and where Huet's zipper comes from.

$$
\begin{aligned}
&= \partial(1 + x \cdot \mathrm{BT}^2(x)) \\
&= 0 + \mathrm{BT}^2(x) + 2 \cdot x \cdot \mathrm{BT}(x) \cdot \partial\mathrm{BT}(x) \\
&= \mathrm{BT}^2(x)/(1 - 2 \cdot x \cdot \mathrm{BT}(x)) \\
&= \mathrm{BT}^2(x) \cdot \mathrm{List}(2 \cdot x \cdot \mathrm{BT}(x)) \\
&= \mathrm{BT}^2(x) \cdot \mathrm{Zipper}_2(x)
\end{aligned}
$$

Bags

- Let's recall the definition of a List after our Taylor series:

$$\mathrm{List}(x) = 1 + x + x^2 + x^3 + \cdots$$

- What if we try and forget the order of our list elements? Remember there's n! ways to order n elements. Then by identifying the combinations, then instead of finding lists with n elements, we get bags (unordered lists) or n elements.

$$\mathrm{Bag}(x) = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^k}{k!} + \cdots = e^x$$

Bags, continued

- Obviously this is very literally not true. The third term of Bag(Bool) would be $2^3/3!$, which is 4/3, and there are not 4/3 3-bool bags.
  - We formally are moving from polynomial functors to analytic functors.
- Although e^x is literally not correct, it does satisfy a lot of properties that we would like in bags:
  - Remember $e^x * e^y = e^{x+y}$. This is analogous to me decomposing a bag of Ints and Bools into two bags – you can still recombine them into one bag. You can't do the same for Lists – you lose the information about the order if you split up the list.
  - The most famous property of e^x is that it is its own derivative. What happens when you take a hole in a bag? You just get another bag. So it does match our version of derivative.

Other Taylor expansions

- Now that we have all this analytic machinery, it's natural to try and use our tools on other Taylor series. Consider the following:

$$\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \qquad \sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots$$

- Note that cosh is the type of bags with even number of elements and sinh is the type of bags with an odd number of elements. So to answer the question from the beginning...
- **cosh(List(bool)) is the type of bags of even number lists of booleans.**
- It's easy to check the normal trigonometric identities hold, even in our type-driven construction.

How did we do all of this?

- We started by stating types form a semiring because they support addition and multiplication.
- From there we did a lot of crazy stuff including subtraction and division and even analysis!
- Nobody here (including me) understands why this all works. But if you are interested, here is the outline:
  - The additive structure can be extended to an abelian group by the Grothendieck group
  - We can then build the field of fractions.
  - In this field, the Puiseux Series and Levi-Civita field allow us to do a limited form of “analysis” without topology or limits
  - I have also heard this is very similar to combinatorial species

Conclusion

- This presentation covered a lot. You really only need to know section I (and you really should, product and sum types are a very useful way to think about things).
- That being said, the math in sections II and III are a very fun exploration of the math we can do with types. I've linked some other resources that I drew upon when making this presentation:
  - The Algebra of Algebraic Data Types (accessible article)
  - Rational Types (an accessible paper on what negative and rational types mean)
  - Conor McBride's blog (more advanced but real papers)
  - Algebra on Data Types (very dense but good)