The Math of Types

Ryan Orendorff

June 13th, 2015

What we will talk about today

Time permitting, I would like to cover the following topics.

- The math of regular data types.
- What a zipper is, and why you might want one.
- Dissection and delimited continuations (as time permits).

The Math of Regular Data Types

What is an Algebra?

In the most general sense, "algebra is the study of mathematical symbols and the rules for manipulating these symbols".[1]

Algebraic structures provide a way for us to "bolt on" manipulation powers onto a set of symbols.

In this talk, we will consider symbols that are both numbers and types.

[1] https://en.wikipedia.org/wiki/Algebra

Magmas

A magma is a binary function $\otimes$ that takes two symbols and produces a third from the same set of symbols.[2]

---

[2] the fact it returns something from the same set makes the operation *closed*

Magmas

A magma is a binary function $\otimes$ that takes two symbols and produces a third from the same set of symbols.[2]

For example, if the symbols we are dealing with are numbers, then we can consider $\cdot$ to be a binary operator that performs this property.

[2] the fact it returns something from the same set makes the operation *closed*

Magmas

A magma is a binary function $\otimes$ that takes two symbols and produces a third from the same set of symbols.[2]

For example, if the symbols we are dealing with are numbers, then we can consider $\cdot$ to be a binary operator that performs this property.

On types, one operation we can do to combine two types is to create a pair, which is itself a type.

```
data (,) a b = (a, b)
```

[2] the fact it returns something from the same set makes the operation *closed*

Monoids

Monoids are similar to magmas. They need a binary operator, but it has to have an identity element such that

$$a \otimes 0 = 0 \otimes a = a$$

Monoids

Monoids are similar to magmas. They need a binary operator, but it has to have an identity element such that

$$a \otimes 0 = 0 \otimes a = a$$

Our multiplication on numbers still follows this rule if we choose 1 as the identity element.

$$a \cdot 1 = a$$

Monoids

Monoids are similar to magmas. They need a binary operator, but it has to have an identity element such that

$$a \otimes 0 = 0 \otimes a = a$$

Our multiplication on numbers still follows this rule if we choose 1 as the identity element.

$$a \cdot 1 = a$$

On types, do we have the same property? Is there a **`One`**?

$$(a, One) \simeq a$$

Sidenote: This *up to isomorphism* thing

When we talk about something "up to isomorphism" for the types, we mean that there is a way to convert between the types. Specifically, if

```
f :: A → B ; g :: B → A
```

then

```
f . g = id; g . f = id
```

Sidenote: This *up to isomorphism* thing

When we talk about something "up to isomorphism" for the types, we mean that there is a way to convert between the types. Specifically, if

```
f :: A → B ; g :: B → A
```

then

```
f . g = id; g . f = id
```

For example, we define

```
type A = (Int, (String, Int))
type B = ((Int, String), Int)

f (a, (b, c)) = ((a, b), c)
g ((a, b), c) = (a, (b, c))
```

We will use $\simeq$ to mean "equal by isomorphism".

So what is `OneType`?

We need a value that we can always pair with some **a** that does not carry any information. What does this mean? I should be able to write these functions.

```
f :: (a, One) → a
g :: a → (a, One)
```

We need a type where there is only one way to make a value of it.

So what is `OneType`?

We need a value that we can always pair with some `a` that does not carry any information. What does this mean? I should be able to write these functions.

```haskell
f :: (a, One) -> a
g :: a -> (a, One)
```

We need a type where there is only one way to make a value of it.

How about

```haskell
data One = One -- Also written as data () = ()
               --               or data Unit = Unit
```

So what is **OneType**?

We need a value that we can always pair with some **a** that does not carry any information. What does this mean? I should be able to write these functions.

```haskell
f :: (a, One) -> a
g :: a -> (a, One)
```

We need a type where there is only one way to make a value of it.

How about

```haskell
data One = One -- Also written as data () = ()
               --                  or data Unit = Unit

f (a, _) = a
g a = (a, One)
```

What we have so far

Operations	Numbers	Types
$\otimes$	$\cdot$	`(,)`
$\otimes$ associativity	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	`(a, (b, c))` $\simeq$ `((a, b), c)`
$\otimes$ identity	1	`One` or `()`
$\otimes$ identity law	$a \cdot 1 = a$	`(a, One)` $\simeq$ `a`

We didn't talk about associativity, but monoids must have it as well.

What we have so far

Operations	Numbers	Types
$\otimes$	$\cdot$	`(,)`
$\otimes$ associativity	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	`(a, (b, c))` $\simeq$ `((a, b), c)`
$\otimes$ identity	1	`One` or `()`
$\otimes$ identity law	$a \cdot 1 = a$	`(a, One)` $\simeq$ `a`

We didn't talk about associativity, but monoids must have it as well.

But I think we forgot something

What about addition?

We can certainly add numbers as well, with the following properties

$$\begin{aligned} a + 0 &= a \\ a + (b + c) &= (a + b) + c \end{aligned}$$

What about addition?

We can certainly add numbers as well, with the following properties

$$
\begin{aligned}
a + 0 &= a \\
a + (b + c) &= (a + b) + c
\end{aligned}
$$

Numbers form a monoid using the **+** operator.

Can we do sums with types?

What about addition?

We can certainly add numbers as well, with the following properties

$$a + 0 = a$$
$$a + (b + c) = (a + b) + c$$

Numbers form a monoid using the `+` operator.

Can we do sums with types?

What about

```
data Either a b = Left a | Right b
```

Addition on Types

Let's see if `Either` makes sense over the associativity property.

```
Either A (Either B C) == Either (Either A B) C
```

Addition on Types

Let's see if `Either` makes sense over the associativity property.

```
Either A (Either B C) == Either (Either A B) C
```

```
f (Left a) = Left (Left a)
f (Right (Left b)) = Left (Right b)
f (Right (Right c)) = Right c
```

and `g` is defined similarly.

What is the identity element for addition?

For numbers, it is 0. For types, we need something that can never happen.

What is the identity element for addition?

For numbers, it is 0. For types, we need something that can never happen.

```haskell
data Void -- no data constructors
```

What is the identity element for addition?

For numbers, it is 0. For types, we need something that can never happen.

```haskell
data Void -- no data constructors
```

This means, for `Either a Void`, it is impossible to call the `Right` constructor.

What is the identity element for addition?

For numbers, it is 0. For types, we need something that can never happen.

```
data Void -- no data constructors
```

This means, for `Either a Void`, it is impossible to call the `Right` constructor.

From this we can easily show `Either a Void` $\simeq$ `a`

The combined total of our investigation

Operations	Numbers	Types
$\otimes$	$\cdot$	`(,)`
$\otimes$ associativity	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	`(a, (b, c))` $\simeq$ `((a, b), c)`
$\otimes$ identity	1	`One` or `()`
$\otimes$ identity law	$a \cdot 1 = a$	`(a, One)` $\simeq$ `a`
$\oplus$	$+$	`Either`
$\oplus$ associativity	$a + (b + c) = (a + b) + c$	`Either A (Either B C)` $\simeq$
		`Either (Either A B) C`
$\oplus$ identity	0	`Void`
$\oplus$ identity law	$a \cdot 0 = a$	`Either a Void` $\simeq$ `a`

And there are a few other laws

All of these laws are followed by numbers and types. This makes both numbers and types an algebraic structure called a semiring.

Operations	Numbers	Types
$\otimes$ commute	$a \cdot b = b \cdot a$	`(a, b)` $\simeq$ `(b, a)`
$\oplus$ commute	$a + b = b + a$	`Either A B` $\simeq$ `Either B A`
$\oplus$ implied equality	$a + b = a + c \implies b = c$	`Either A B` $\simeq$ `Either A C` $\implies$ `B` $\simeq$ `C`
Left distributive	$(a + b) \cdot c = a \cdot c + b \cdot c$	`(Either A B, C)` $\simeq$ `Either (A, C) (B, C)`
Zero is not One	$0 \neq 1$	`Void` $\not\simeq$ `One`

How can we use this?

Using the semiring operations, we can start to build up terms programmatically.

For example, take the `Maybe` type.

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

How can we use this?

Using the semiring operations, we can start to build up terms programmatically.

For example, take the `Maybe` type.

```
data Maybe a = Nothing | Just a

type Maybe a = Either One a
```

How can we use this?

Using the semiring operations, we can start to build up terms programmatically.

For example, take the `Maybe` type.

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

```
type Maybe a = Either One a
```

In number this is $Maybe(x) = 1 + x$

Making the algebra easier to see

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DeriveFunctor #-}

newtype K a x = K a deriving (Show, Functor)
newtype Id x = Id x deriving (Show, Functor)
newtype (p :*: q) x = Prod (p x, q x) deriving (Show, Functor)
data (p :+: q) x = L (p x) | R (q x) deriving (Show, Functor)

type One = K ()

one = K ()
```

The paper “Algebra of Programming” (Bird, de Moor 1997) has a much more in depth coverage.[3]

[3] Copied directly from the Clowns and Jokers paper

Recreating Maybe

```
type Maybe = One :+: Id

just :: x → Maybe x
just x = R (Id x)

nothing :: Maybe x
nothing = L one
```

Recreating Maybe

```
type Maybe = One :+: Id

just :: x → Maybe x
just x = R (Id x)

nothing :: Maybe x
nothing = L one

fmap (+2) nothing == L one
```

Recreating Maybe

```haskell
type Maybe = One :+: Id

just :: x -> Maybe x
just x = R (Id x)

nothing :: Maybe x
nothing = L one

fmap (+2) nothing == L one

fmap (+2) (just 5) == R (Id 7)
```

Notice no functor instance was declared for `Maybe`.

Zippers

Motivation: Updating lists has poor performance

Say you want to update one element of a list. Here is one way to do that.

```haskell
-- Insanely partial
update :: Int -> (a -> a) -> [a] -> [a]
update index _ [] = error "Hey, I need something to edit dummy!"
update index f (x:xs) = if index == 0
                        then f x : xs
                        else x : update (index - 1) f xs
```

Motivation: Updating lists has poor performance

Say you want to update one element of a list. Here is one way to do that.

```
-- Insanely partial
update :: Int -> (a -> a) -> [a] -> [a]
update index _ [] = error "Hey, I need something to edit dummy!"
update index f (x:xs) = if index == 0
                        then f x : xs
                        else x : update (index - 1) f xs
```

As an example, lets update an element of a simple list.

```
update 2 (10 `const`) list == [1, 2, 10, 4, 5]
```

Where is the poor performance?

Say we start with some list `l` with six elements.

```
    index
    0   1   2   3   4   5
l = a : b : c : d : e : f : []
```

Now let's update the third element using some update function `f`.

Where is the poor performance?

Say we start with some list `l` with six elements.

```
    index
    0   1   2   3   4   5
l = a : b : c : d : e : f : []
_                   ^_______^
_                       └___________┐
update 3 f l == a : b : c : f d : _ |
```

We now have a bunch of copies of elements *we didn't touch!*

The performance problem is a compounding problem

If we made many updates near the same location, then we keep allocating new nodes.

```
    index
    0   1   2   3   4   5
l = a : b : c : d : e : f : []
--                  ^-----^
--                     └------------┐
m = update 3 f l == a : b : c : f d : -- |
--                                       └--┐
--                                          |
n = update 3 g m == a : b : c : g (f d) : -- |

-- etc.
```

Note that this not just poor space usage but poor *time* as well. Each update is $O(n)$.

What if we “pause” our traversal?

The main problem we have is that we keep traversing the list over and over, making new lists every time.

```
m = update i f l — maybe makes an edit at the end
update i f m — update is in same spot i, and we traverse m to the
```

What happens if we instead we separate the list before and after the index `i`?

```
type PauseList a = ([a], [a])
```

Converting to a PauseList

Here are some helper functions to convert from/to a **PauseList** from a **List**.

```haskell
toPause :: Int → [a] → PauseList a
toPause i xs = (reverse . take i $ xs, drop i xs)

fromPause :: PauseList a → [a]
fromPause (prior, curnext) = reverse prior ++ curnext

current :: PauseList a → a
current (_, x:_) = x
```

Converting to a PauseList

Here are some helper functions to convert from/to a **PauseList** from a **List**.

```
toPause :: Int -> [a] -> PauseList a
toPause i xs = (reverse . take i $ xs, drop i xs)

fromPause :: PauseList a -> [a]
fromPause (prior, curnext) = reverse prior ++ curnext

current :: PauseList a -> a
current (_, x:_) = x

p = toPause 2 [0, 1, 2, 3, 4, 5] == ([2, 1], [3, 4, 5])
current p == 3
```

Moving in a **PauseList**

And some ways in which to move around a **PauseList**

```
forward :: PauseList a -> PauseList a
forward (prior, current : next) = (current:prior, next)

backward :: PauseList a -> PauseList a
backward (lastcur:prior, curnext) = (prior, lastcur:curnext)
```

Moving in a PauseList

And some ways in which to move around a **PauseList**

```haskell
forward :: PauseList a -> PauseList a
forward (prior, current : next) = (current:prior, next)

backward :: PauseList a -> PauseList a
backward (lastcur:prior, curnext) = (prior, lastcur:curnext)

p = toPause 2 [0, 1, 2, 3, 4, 5] == ([2, 1], [3, 4, 5])
forward p = ([3, 2, 1], [4, 5])
```

Updating a PauseList

Now let's look at the **updatePause** function to update the current value.

```
updatePause :: (a → a) → PauseList a → PauseList a
updatePause f (prior, current : next) = (prior, f current : next)
```

Here, only 1 node is being created, and it is the node being modified.

`udpatePause` pictorally

We'll use this notation for a `PauseList` with the cursor on the third element.

```
pl =  a ← b ← _c_ → d → e → []
```

`udpatePause` pictorally

We'll use this notation for a **`PauseList`** with the cursor on the third element.

```
 pl =  a ← b ← _c_ → d → e → []
—          ^             ^
—          └─────┐    └─┐
updatePause f pl —    └ f c ┘
```

Now local updates are a quite efficient $O(1)$.

Going forward and backward also has a constant cost

Similarly, moving the current position has a constant cost.

```
forward :: PauseList a → PauseList a
forward (prior, current : next) = (current:prior, next)

 pl =  a ← b ← _c_ → d → e → []
—          ^                ^
—          └————┐           └┐
forward pl —        c ← _d_  |
```

What we have created is called a Zipper or One Hole Context

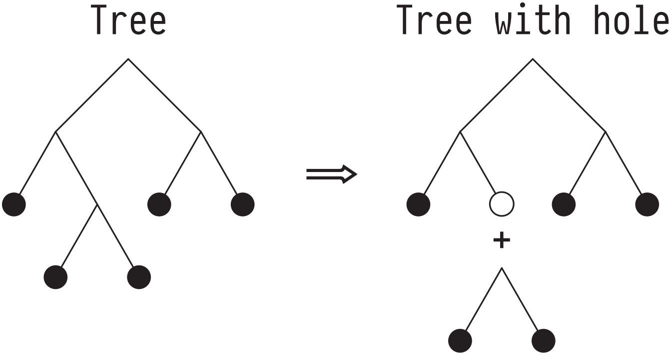
The generic “pausing” structure is often known as a zipper, or as a one hole context. Here we will be focusing on the one hole context nomenclature.

Examples of zippers

- ZipperFS
- Xmonad

One-hole types

We can think of one-hole types as "focusing" on an element.
Another canonical example is a tree.



Let's do a simple example

What is the context for a product type?

`(x, x)` — `sometimes called` x^2

[4] here ∘ will represent the focus

Let’s do a simple example

What is the context for a product type?

`(x, x)` — `sometimes called` x^2

Well we either be focusing on the left side[4]

$(\boxed{\circ}, x)$

[4] here ∘ will represent the focus

Let's do a simple example

What is the context for a product type?

`(x, x) — sometimes called` x^2

Well we either be focusing on the left side[4]

`(○, x)`

or the right side.

`(x, ○)`

[4] here ○ will represent the focus

Let's do a simple example

What is the context for a product type?

`(x, x) — sometimes called` x^2

Well we either be focusing on the left side[4]

$(\boxed{\circ}, x)$

or the right side.

$(x, \boxed{\circ})$

We can represent this as `Either (One, x) (x, One)` (sometimes written as $2x$).

[4] here $\circ$ will represent the focus

And now a triple

What is the context for a triple?

`(x, x, x)` — `sometimes called` x^3

[5] here ○ will represent the focus

And now a triple

What is the context for a triple?

`(x, x, x)` — `sometimes called` x^3

Well we either be focusing on the left side[5]

`(○, x, x)`

or the middle

`(x, ○, x)`

or the right side

`(x, x, ○)`

[5] here ○ will represent the focus

And now a triple

What is the context for a triple?

`(x, x, x)` — `sometimes called` x^3

Well we either be focusing on the left side[5]

`(○, x, x)`

or the middle

`(x, ○, x)`

or the right side

`(x, x, ○)`

We can represent this as `Either (Either (One, x, x) (x, One, x)) (x, x, One)` (sometimes written as $3x^2$).

[5] here ○ will represent the focus

And the context for One?

Can we have a one hole context for the `One` type?

```
data One = One
```

And the context for One?

Can we have a one hole context for the **One** type?

```
data One = One
```

Well we have nothing to take the context of (an **x**), so we can represent the context as impossible.

```
data Void
```

Enumeration Context

Actually, this trick can be done for any simple enumeration.

```
-- Some enumeration
data Cards = Hearts | Spades | Clubs |Diamonds -- 4
```

Enumeration Context

Actually, this trick can be done for any simple enumeration.

```
-- Some enumeration
data Cards = Hearts | Spades | Clubs |Diamonds -- 4

-- And the context with respect to some variable
data Void
```

Context for addition

Say we have the data type

```
data Crafty x = Left x | Right (x, x)
```

What is the one hold context for this?

Context for addition

Say we have the data type

```
data Crafty x = Left x | Right (x, x)
```

What is the one hold context for this?

Well we could either have a hole in the left side, or a hole in one of two positions on the right side.

```
Either One (Either (One, x) (x, One))
```

A table of what we have so far

Type	Number	Context	Number
One	1	Void	0
Cards	4	Void	0
(x, x)	x^2	`(x, One) :+: (One, x)`	$2x$
(x, x, x)	x^3	`(One, x, x) :+: (x, One, x) :+: (x, x, One)`	$3x^2$
Crafty	$x^2 + x$	`(x, One) :+: (One, x) :+: One`	$2x + 1$

A table of what we have so far

Type	Number	Context	Number
One	1	Void	0
Cards	4	Void	0
(x, x)	x^2	`(x, One) :+: (One, x)`	$2x$
(x, x, x)	x^3	`(One, x, x) :+: (x, One, x) :+: (x, x, One)`	$3x^2$
Crafty	$x^2 + x$	`(x, One) :+: (One, x) :+: One`	$2x + 1$

That looks an awful lot like differentiation.

What else is a differentiation

What is the differentation of `[a]`?

What else is a differentiation

What is the differentation of `[a]`?

([a], [a])

What have we learned thus far?

We have learned

- how to do algebra on types.
- What a zipper is
- That taking the derivative of a type gives us back a zipper.

There are other differentiation laws such as the chain rule for composition and differentiating a fixed point type.

Dissection

An example use of a zipper

Let's look at a binary tree.[6]

```
data Expr = Val Int | Add Expr Expr

eval :: Expr -> Int
eval (Val x) = x
eval (Add e1 e2) eval e1 + eval e2
```

[6]this example stolen wholesale from the Clowns and Jokers paper.

Trying a tail recursive version

```
type Stack = [Expr :+: Int]

eval :: Expr -> Int
eval e = load e []

load :: Expr -> Stack -> Int
load (Val i) stk = unload i stk
load (Add e1 e2_ = load e1 (Left e2 : stk)

unload :: Int -> Stack -> Int
unload v [] = v
unload v1 (Left e2: stk) = load e2 (Right v1 : stk)
unload v2 (Right v1 : stk) = unload (v1 + v2) stk
```

Eval can be implemented generically two ways

- Differentiating a data type (the Conor McBride method)
- Delimited continuations to pause a traversal (the Oleg Kiselyov method)

Examples

- Folding a tree, finding its maximum path weight, height, etc
- Generating a tree (ex: a Stern-Brocot tree)
- Inverting a tree to perform some action at the leaves (ex: hasSuffix)

Quote

"But if there is a message for programmers and programming language designers, it is this: the miserablist position that types only exist to police errors is thankfully no longer sustainable, once we start writing programs like this [generic zippers]. By permitting calculations of types and from types, we discover what programs we can have, just for the price of structuring our data. What joy!"

– Conor McBride

References

All article names are clickable links.

- Clowns to the Left of me, Jokers to the Right by Conor McBride (the POPL version).
- Organizing Numerical Theories using Axiomatic Type Classes by Pawrence Paulson
- The Algebra of Albebraic Data Types by Chris Taylor (written version 1, 2, and 3)
- Category Theory for Programmers by Bartosz Milewski
- Conquering Folds by Edward Kmett
- Zippers Part 1, 2, 3 by Pavel Panchekha
- Polynomial Functors Constrainted by Regular Expressions by Dan Piponi and Brent Yorgey