Functional Algebra for Middle School Students

Chad Brewbaker
DataCulture LLC

May 27, 2016

How can we get functional programming into universal education?

What language did we use before COBOL and ALGOL?

How can we refactor on a napkin without an IDE?

How can we prove our code is correct?

How can we easily get programming in the Jr. High curriculum?
(Algebra)
What language did we use before COBOL and ALGOL?
(Algebra)
How can we refactor on a napkin without an IDE?
(Algebra)
How can we prove our code is correct?
(Algebra)

1) Ignore fancy terminology.

2) Use sums, products, and exponents.

3) Teach Jr. High Algebra instructors just enough about type signatures.

The Haskell language

Looks like Algebra.

Easy to reason about. No rando database calls in a function without IO warning.

Maximum flexibility for abstract design.

The Ruby language

Lots of existing projects for kids.

Play-dough on top of C.

Maximum flexibility for human tinkering.

The C language

Loosely typed machine instructions.

Maximum flexibility for machines.

Function notation: Algebra, Haskell, C, Ruby

$f(\mathbb{Z}, \mathbb{Z}) \to \text{String}$

```
f :: (Int, Int) -> String
```

```
String f(int a, int b);
```

Longform functions in Ruby

```
def f(a,b)
  #function body
end
```

Short form functions in Ruby

```
f = ->(a,b){
   #function body
}
```

Sum: A or B

$$A + B$$

```
data X = A | B
```

Sally's drink is either Coffee or Tea.

```
type Drink = Coffee | Tea
```

Sum: A or B

$$A + B$$

C has enumeration types

```
enum X { A, B };
```

Ruby has enumerable types

```
x = Set.new [a,b]
```

Product: A and B

$$A \times B$$

Haskell has product types, also known as tuples

```
type X = (A,B)
```

Sally's dinner consists of a fruit and a vegetable.

```
type Dinner = (Fruit, Vegetable)
```

Product: A and B

$$A \times B$$

C has structs.

```
struct X{ A a; B b;};
```

Product: A and B

$$A \times B$$

Ruby has Array types.

```
x = [a, b]
```

Ruby classes can be used as product types.

```
class Product
  @a = A.new()
  @b = B.new()
end
x = Product.new()
```

Exponential: From A to B

$$B^A$$

A is called the domain or input type. B is called co-domain or return type.

```
f :: A -> B
```

Sally chops a carrot.

```
chop :: Carrot -> DicedCarrot
```

Terminology for simple values.

$\sqrt{-1}$ Undefined (Irrational)

```
0 Void
1 Unit, also ()
2 Bool
3 Tri
4 Quad
```

Constants are functions with the Unit as an argument.

$$a = a^1$$

```
foo :: A

bar :: () -> A
```

Tuples vs tuple lookups

$$a \times a = a^2$$

```
foo :: (A,A)
```

```
bar :: Bool -> A
```

Interpret *bar* as choice between left A and right A.

Void behaves like zero

$$1 = a^0$$

```
foo :: Unit

bar :: Void -> A
```

$$a \times 0 = 0$$

```
foo :: (A, Void)

bar :: Void
```

Void behaves like zero

$$a + 0 = a$$

```
foo :: A | Void

bar :: A
```

Derivative(N) is removing one element from N

$$\frac{d}{dx}(ax^n) = a \times n \times x^{n-1}$$

```
data N = M | 1

foo ::   derivitaveN( a, N -> x)

bar :: (a, N, M -> X)
```

Curry. We can pass argument lists or chain functions.

$$(a^m)^n = a^{mn}$$

```
curryFoo :: n -> (m -> a)

curryBar :: (n,m) -> a
```

```ruby
foo = ->(n,m){n+m}
p foo.call("Thing1","Thing2")

bar = foo.curry.("Thing1")
p bar.call("Thing2")
```

```
"Thing1Thing2"
"Thing1Thing2"
```

Co-curry. Multiple function calls or return a tuple.

$$a^m b^m = (ab)^m$$

```
cocurryFoo :: (m -> a, m -> b)

cocurryBar :: m -> (a,b)
```

```ruby
foo = [ ->(m){m+m}, ->(m){m}]
p [foo[0].call("Thing"), foo[1].call("Thing")]

bar = ->(m){ [m+m, m]}
p bar.call("Thing")
```

```
["ThingThing", "Thing"]
["ThingThing", "Thing"]
```

Splitting function inputs

$$a^m a^n = a^{m+n}$$

```
data B = M | N

foo :: (M -> a, N -> a)

bar :: B -> a
```

```
foo = [->(even){even}, ->(odd){2*odd +1} ]
p [foo[0].call(2), foo[1].call(3)]

bar = ->(x){
 if( x%2 == 0)
   return x
 else
   return 2*x + 1
 end
}
p [bar.call(2), bar.call(3)]
```

```
[2, 7]
[2, 7]
```

Ignoring some input or shrinking function input

$$a^m \div a^n = a^{m-n}$$

```
data M = B | N

f :: (M -> a) remove (N -> a)

f' :: B -> a
```

```ruby
foo = ->(x){
 if( x%2 == 0)
    p "ERROR we are ignoring evens"
 end
 return 2*x +1
}
p foo.call(3)

bar = ->(x){2*x + 1}
p bar.call(3)
```

```
7
7
```

Suppressing some outputs or shrinking the output type

$$a^m \div b^m = (\frac{a}{b})^m$$

```
data A = N | B

f :: (m -> A) remove (m -> B)

f' :: m -> N
```

Fermat's Little Theorem
$a^n - a$ is divisible by n when n is prime.

$$a^n - a = n \times k$$

$$a^n = n \times k + a$$

```
foo :: N -> A

bar :: (N, K) | A
```

This refactoring exists whenever N is a prime size set.

Fermat's Last Theorem

$x^n + y^n = z^n$ has no solutions for $n > 2$.

```
foo :: (N -> X) | (N -> Y)

bar :: (N ->Z)
```

foo $\neq$ bar when N is greater than size two.

Binomial Theorem.

Refactor between product of sums and sum of products.

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

$$(a+b)^2 = a^2 + 2ab + b^2$$

```
foo :: Bool -> (A | B)

bar :: Bool -> A
     | (Bool, A, B)
     | Bool -> B
baz :: (A,A) | ((A,B) | (B,A)) | (B,B)
```

Tips on how to read Algebra as a programmer.

All the functions from N to N.

$$N^N$$

All reorderings of N.

$$n! = n * (n-1) * ... * 2 * 1$$

Let's ignore some stuff that is type X.

$$\frac{1}{x}$$

Let's ignore reordering.

$$\frac{1}{n!}$$

I'm about to give a stream of things.

$$\sum$$

I've got an arbitrarily large tuple.

$$\Pi$$

Moving forward.

Teach Jr. High Algebra teachers basic type signatures.

Functions are exponents. More practice problems on towers of exponents.

“Abstract” algebra can be obtuse. Be descriptive with concrete examples. Less jargon.

My research project Endoscope. Inspect function structure. Don’t rely on intuition.

BONUS SLIDES

Binary tree.

Empty tree or a parent node and two children.

Tree A = EmptyTree + (Parent A ,Left A, Right A)

$T_a = 1 + aT_a{}^2$

$aT_a{}^2 - T_a + 1 = 0$

Solve with quadratic formula.

What the heck do we do with this?

$$T_a = \frac{1 - \sqrt{1 - 4a}}{2a}$$

Wolfram Alpha, "Series [[1 - Sqrt[1-4a]] / [2a]] at 0"

$$1 + a + 2a^2 + 5a^3 + 14a^4 + 132a^6...$$

```
data SixTree A = (Void, Void -> a)   | (Unit, Unit -> A)
    | (Quint, Tri -> A) | (Fourteen, Quad -> A)
    | (OneThrityTwo, Hex -> A)
```

Monads are foreign because we aren't used to Algebra with towers of exponents. m is usually a container or operating system call.

Sequentially compose two items in the m context. Pipe the return of the first into the second.
$(>>=) = {m_b}^{{m_b}^a m_a}$

Sequentially compose two items in the m context. Do not pipe the output of the first to the second.
$(>>) = {m_b}^{m_b m_a}$

Put a in the m context.
return $= {m_a}^a$

```
import System.Environment as SE
import System.Process as SP

printPath = SE.getExecutablePath >>= print
sentence = "say␣Two␣plus␣one␣equals␣three"
twoPlusOne = SP.system sentence
             >> print (2+1)

saywhat :: String -> String
saywhat x = "say␣" ++  x
sayit x = SP.system $ saywhat x

sayExePath = SE.getExecutablePath
             >>= \x -> sayit x
```

Bibliography

Haskell Curry(1934), “Functionality in Combinatory Logic”

Eric G Wagner(1986), “Categories, Data Types, and Imperative Languages”

Douglas McIlroy(1998), “Power Series, Power Serious”

Chris Taylor(2013), “The Algebra of Algebraic Data Types”