Programming with Jr. High Algebra

Chad Brewbaker

May 5, 2016

How can we get programming in the Jr. High curriculum?

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 Haskell.

The Haskell language

Advised by Simon Peyton Jones of Microsoft Research.

Looks like Algebra.

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

Function notation: Algebra, Haskell, C#

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

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

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

Sum: A or B

$$A + B$$

```
data X = A | B
f :: X -> Either A B
```

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

Sally's drink is either Coffee or Tea.

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

Product: A and B

$$A \times B$$

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

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

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

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

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
```

```
B f( A a); //C
```

Sally chops a carrot.

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

Exponential: B^A

Example from JEG's RailsConf2016 talk

```
Integer fib(Integer a){
    if (a == 0 || a == 1)
        return 1;
    return fib(a-1) + fib(a-2);
}
```

What is the runtime of fib(n)?

Exponential: B^A

Focus on this line

```
return fib(a-1) + fib(a-2);
```

How many branches? Bool
How deep is each branch? $\approx A$

```
fibExecutionUnit :: A -> Bool
```

$$O(2^A)$$

Terminology for simple values.

$\sqrt{-1}$ Undefined

0 Void
1 Unit
2 Bool
3 Tri
4 Quad

Constants are functions with the Unit as an argument.

$$a = a^1$$

```
f  ::  A
f' ::  () -> A
```

Tuples vs tuple lookups

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

```
f  :: (A,A)
f' :: Bool -> A  -- Choose the left or right A
```

Void behaves like zero

$$1 = a^0$$

```
f  :: Unit
f' :: Void -> A
```

$$a \times 0 = 0$$

```
g  :: (A, Void)
g' :: Void
```

$$a + 0 = a$$

```
h  :: A | Void
h' :: 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
f  ::   (a, N -> x)
f' ::  (a, N, M -> X)
```

(a, which element we took out of N, the new function)

Curry. We can pass argument lists or chain functions.

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

```
curry  :: n -> (m -> a)
curry' :: (n,m) -> a
```

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

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

```
f  :: (m -> a, m -> b)
f' :: m -> (a, b)
```

Splitting function inputs

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

```
data B = M | N
f :: (M -> a, N -> a)
f' :: B -> a
```

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
```

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.

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

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

```
f  :: N -> A
f' :: (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 $n > 2$.

```
f  ::  (N -> X)  |  (N -> Y)
f' ::  (N ->Z)
```

This refactoring never exists when N is greater than size two.

Binomial Theorem.
Refactor between product of sums and sum of products.

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

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

```
f  :: Bool -> (A | B)
f' :: Bool -> A
        | (Bool, A, B)
        | Bool -> B
f'':: (A,A) | ((A,B) | (B,A)) | (B,B)
```

Binomial Theorem

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

```
f  ::  Tri -> (A | B)
f' ::   Tri -> A
      | (Tri, Bool -> A, B)
      | (Tri, A, Bool -> B)
      | Tri -> B
f'' :: (A,A,A)
      | ( (A,A,B) | (A,B,A) | (B,A,A) )
      | ( (A,B,B) | (A,B,B) | (B,A,B) )
      | (B,B,B)
```

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)
```

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.

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

Let's remove one element from x in $x \to y$.

$$f^{'}(x)$$

Let's remove two elements from x in $x \to y$.

$$f^{''}(x)$$

Let's ignore reordering.

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

I'm about to give a stream of things.

$$\sum$$

I've got an arbitrarily large tuple.

$$\Pi$$

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
```

Can we automate how we write code?
I give the compiler two data types A and B.
The compiler searches for functions from A to B that typecheck.
I give the compiler unit test hints so it avoids silly code paths.
The compiler gives me hints on where it is getting stuck.

This tool is the Microsoft LEAN project.

My take on the future of automated programming.

Jr. High Algebra is a good language. LEAN jumped the shark.

We have free symbolic Alegebra packages like SymPy.

Where SymPy fails we have Microsoft's Z3 solver.

10x humans ask questions and give advice. Calculations are for computers.

Stay tuned for talk later this month "Z3 and me".

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”