Good Math, Bad Math

The Genius of Alonzo Church (rerun)

Posted by **Mark C. Chu-Carroll** on August 30, 2006

(1) I'm on vacation this week, so I'm posting reruns of some of the better articles from when Goodmath/Badmath was on Blogger. Todays is a combination of two short posts on numbers and control booleans in λ calculus.

So, now, time to move on to doing interesting stuff with lambda calculus. To start with, for convenience, I'll introduce a bit of syntactic sugar to let us name functions. This will make things easier to read as we get to complicated stuff.

To introduce a *global* function (that is a function that we'll use throughout our lambda calculus introduction without including its declaration in every expression), we'll use a definition like the following:

square ≡ λ x . x × x

This declares a function named "square", whose definition is "*λ x . x×x*". If we had an expression "square 4", the definition above means that it would effectively be treated as if the expression were: "*(λ square . square 4)(λ x . x×x)*".

Numbers in Lambda Calculus

In some of the examples, I used numbers and arithmetic operations. But numbers don't really exist in lambda calculus; all we really have are functions! So we need to invent some way of *creating* numbers using functions. Fortunately, Alonzo Church, the genius who invented the lambda calculus worked out how to do that. His version of numbers-as-functions are called *Church Numerals*.

In Church numerals, all numbers are functions with two parameters:

* Zero ≡ *λ s z . z*
* One ≡ *λ s z . s z*
* Two ≡ *λ s z . s (s z)*
* Three ≡ *λ s z . s (s (s z))*
* Any natural number "n", is represented by a Church numeral which is a function which applies its first parameter to its second parameter "n" times.

A good way of understanding this is to think of "z" as being a a name for a zero-value, and "s" as a name for a successor function. So zero is a function which just returns the "0" value; one is a function which applies the

successor function once to zero; two is a function which applies successor to the successor of zero, etc. It's just the Peano arithmetic definition of numbers transformed into lambda calculus.

But the really cool thing is what comes next. If we want to do addition, x + y, we need to write a function with four parameters; the two numbers to add; and the "s" and "z" values we want in the resulting number:

$$add \equiv *\lambda\ s\ z\ x\ y\ .\ x\ s\ (y\ s\ z)*$$

Let's curry that, to separate the two things that are going on. First, it's taking two parameters which are the two values we need to add; second, it needs to normalize things so that the two values being added end up sharing the same binding of the zero and successor values.

$$add\_curry \equiv \lambda\ x\ y.\ (\lambda\ s\ z\ .\ (x\ s\ (y\ s\ z)))$$

Look at that for a moment; what that says is, to add x and y: create the church numeral "y" using the parameters "s" and "z". Then **apply x** to that new church numeral y. That is: a number is a function *which adds itself to another number*.

Let's look a tad closer, and run through the evaluation of 2 + 3:

$$add\_curry\ (\lambda\ s\ z\ .\ s\ (s\ z))\ (\lambda\ s\ z\ .\ s\ (s\ (s\ z)))$$

To make things easier, let's alpha 2 and 3, so that "2" uses "s2" and "z2", and 3 uses "s3" and "z3";

$$add\_curry\ (\lambda\ s2\ z2\ .\ s2\ (s2\ z2))\ (\lambda\ s3\ z3\ .\ s3\ (s3\ (s3\ z3)))$$

Now, let's do replace "add_curry" with its definition:

$$(\lambda\ x\ y\ .(\lambda\ s\ z.\ (x\ s\ y\ s\ z)))\ (\lambda\ s2\ z2\ .\ s2\ (s2\ z2))\ (\lambda\ s3\ z3\ .\ s3\ (s3\ (s3\ z3)))$$

Now, let's do a beta on add:

$$\lambda\ s\ z\ .\ (\lambda\ s2\ z2\ .\ s2\ (s2\ z2))\ s\ (\lambda\ s3\ z3\ .\ s3\ (s3\ (s3\ z3))\ s\ z)$$

And now let's beta the church numeral for three. This basically just "normalizes" three: it replaces the successor and zero function in the definition of three with the successor and zero functions from the parameters to add.

$$\lambda\ s\ z\ .\ (\lambda\ s2\ z2\ .\ s2\ (s2\ z2))\ s\ (s\ (s\ (s\ z)))$$

Now.. Here comes the really neat part. Beta again, this time on the lambda for two. Look at what we're going to be doing here: two is a function which takes two parameters: a successor function, and zero function. To add two and three, we're using the successor function from add function; and we're using the result of evaluating three *as the value of the zero!* for two:

$$\lambda\ s\ z\ .\ s\ (s\ (s\ (s\ (s\ z))))$$

And we have our result: the church numeral for five!

Choice in Lambda Calculus

Now that we have numbers in our Lambda calculus, there are only two things missing before we can express arbitrary computations: a way of expressing choice, and a way of expressing repetition. So now I'll talk about booleans and choice; and then next post I'll explain repetition and recursion.

We'd like to be able to write choices as if/then/else expressions, like we have in most programming languages. Following the basic pattern of the church numerals, where a number is expressed as a function that adds itself to another number, we'll express true and false values as functions that perform an if-then-else operation on their parameters. These are sometimes called *Church booleans*. (Of course, also invented by Alonzo Church.)

* TRUE ≡ λ t f . t
* FALSE ≡ λ t f . f

So, now we can write an "if" function, whose first parameter is a condition expression, second parameter is the expression to evaluate if the condition was true, and third parameter is the expression to evaluate if the condition is false.

* IfThenElse ≡ *λ cond t f . cond t f*

For the boolean values to be useful, we also need to be able to do the usual logical operations:

* BoolAnd ≡ *λ x y .x y FALSE*
* BoolOr ≡ *λ x y. x TRUE y*
* BoolNot ≡ *λ x . x FALSE TRUE*

Now, let's just walk through those a bit. Let's first take a look at BoolAnd. We'll try evaluating "*BoolAnd TRUE FALSE*":

* Expand the TRUE and FALSE definitions: "*BoolAnd (λ t f . t) (λ t f . f)*"
* Alpha the true and false: "*BoolAnd (λ tt tf . tt) (λ ft ff . ff)*"
* Now, expand BoolAnd: *(λ t f. t f FALSE) (λ tt tf . tt) (λ ft ff . ff)*
* And beta: *(λ tt tf.tt) (λ ft ff. ff) FALSE*
* Beta again: *(λ xf yf . yf)*

And we have the result: *BoolAnd TRUE FALSE = FALSE*. Now let's look at "*BoolAnd FALSE TRUE*":

* *BoolAnd (λ t f . f) (λ t f .t)*
* Alpha: *BoolAnd (λ ft ff . ff) (λ tt tf . tt)*
* Expand BoolAnd: *(λ x y .x y FALSE) (lambda ft ff . ff) (lambda tt tf . tt)*
* Beta: *(λ ft ff . ff) (lambda tt tf . tt) FALSE
* Beta again: FALSE

So *BoolAnd FALSE TRUE = FALSE*. Finally, *BoolAnd TRUE TRUE*:

* Expand the two trues: *BoolAnd (λ t f . t) (λ t f . t)*
* Alpha: *BoolAnd (λ xt xf . xt) (λ yt yf . yt)*

* Expand BoolAnd: *(λ x y . x y FALSE) (λ xt xf . xt) (λ yt yf . yt)*
* Beta: *(λ xt xf . xt) (λ yt yf . yt) FALSE*
* Beta: (λ yt yf . yt)

So *BoolAnd TRUE TRUE = TRUE*.

The other booleans work in much the same way.

So by the genius of Alonzo church, we have *almost* everything we need in order to write programs in λ calculus.
