Lambda Calculus

Peter C. Chapin
Vermont Technical College

November 13, 2015

History

- The lambda calculus was developed by Alonzo Church in the 1930s.
- Church was a contemporary of Turing and was also interested in models of computation.
- Originally developed as a kind of logic. That effort was a failure.
- However, Church realized the lambda calculus could be used as a model of computation.

Basic Syntax

Let t be a *lambda term*. Let X be a countably infinite set of variable symbols.

The syntax of t is as follows.

- $t \to x$, where $x \in X$. A term can be a variable.
- $t \to \lambda x.t$, called *lambda abstraction*.
- $t \to (tt)$, called *application*.

Examples

- $\lambda x.x$
- $\lambda x.(\lambda y.xy)$
- $(\lambda x.(\lambda y.xy))(\lambda x.x)$

Lambda terms are functions: $\lambda x.x$ is the function taking a parameter x and returning what it is given.

α Conversion

You can rename the bound variable (the parameter) to a lambda abstraction.

The renaming must be done uniformly over all instances of that variable in the scope of the abstraction.

$\lambda x.(\lambda y.xy)$

Change x to z

$\lambda z.(\lambda y.zy)$

But...

$\lambda x.(\lambda x.x)x$

Becomes

$\lambda z.(\lambda x.x)z$

β Reduction

The only computation rule.

Simply substitute a function argument into the function's body.

$(\lambda x.(\lambda y.xy))(\lambda x.x)$

$(\lambda y.(\lambda x.x)y)$

$(\lambda y.y)$

Computation stops when no more reductions are possible.

The first expression above evaluates to the identity function.

Church Booleans

It is not obvious how one could do anything useful with this. We need to build up some basic values.

Let T be $(\lambda x.(\lambda y.x))$

Let F be $(\lambda x.(\lambda y.y))$

T is a function taking two arguments and returning the first.

F is a function taking two arguments and returning the second.

Logical Operators

Let **and** be $(\lambda x.(\lambda y.(xy)F))$

Let **or** be $(\lambda x.(\lambda y.(xT)y))$

The expression “T and F” is encoded as

$((\lambda x.(\lambda y.(xy)F))T)F$

Expanding T and F yields

$((\lambda x.(\lambda y.(xy)(\lambda x.(\lambda y.y))))(\lambda x.(\lambda y.x)))(\lambda x.(\lambda y.y))$

Using β reduction, this expression evaluates to F.

Church Numerals

Model natural numbers by repeated function applications.

The function that applies its first argument n times represents the number n.

$c_0 = (\lambda s.(\lambda x.x))$

$c_1 = (\lambda s.(\lambda x.(sx))$

$c_2 = (\lambda s.(\lambda x.s(sx))$

$c_3 = (\lambda s.(\lambda x.s(s(sx)))$

It is now possible to define mathematical operations as lambda terms working on Church numerals.

They are ugly.

Scala Syntax

Functional languages are just syntactic sugar for lambda terms.

Lambda Calculus: $(\lambda x.(\lambda y.x))$

Scala: `(x) => ((y) => x)`

Another example

Scala: `val f = (x:  Int) => x + 1`

Lambda Calculus: $(f = \lambda x.Pxc_1)$ where P is the lambda term for addition and c_1 is the first Church numeral.

It is also possible to define lambda terms to model pairs, conditionals, match expressions, etc.

Recursion?

Turing completeness requires an ability to compute forever.

Won't the β reduction process always end?

No!

$(\lambda x.(xx))(\lambda x.(xx))$

This reduces to itself; an infinite loop.

More complex terms allow for eventual termination. Recursive functions are possible.

Turing Complete

Lambda calculus is Turing complete.

- Simulation of β reduction on a Turing machine is obvious enough. Store the lambda term on the tape and progressively rewrite it.
- Simulation of a Turing machine with the Lambda Calculus is less obvious. The Turing tape can be simulated using function argument values in nested calls.
- Infinite recursive functions thus support the unbounded Turing tape.

This is provable: The Lambda Calculus is computationally complete. There does not exist an algorithm that can't be represented by it.

Reality Check

- Real computers are like Turing machines. (Memory is the Turing tape, CPU is the state machine).

- Imperative languages use the Turing machine model. (Tape is rewritten with updates as the program executes).

- Thus imperative languages are a more natural fit to the hardware. *Faster!*

BUT... Clever optimization techniques allow modern functional compilers to produce reasonably fast code.

Difference not that great in practice (today).

Both Approaches Valuable

Functional approach good in some situations.

- Lack of mutable state makes it easier to write bug-free code.
- Mathematical basis makes reasoning about programs easier.
- Lack of mutable state makes parallelizing programs easier.

Functional languages typically "impure" to some degree to deal with I/O (external interactions).

BUT... modern imperative languages typically have some functional features as well.

You will see these ideas in the future!