The Lambda Calculus

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

A Quote to Start the Day

*You realize that **everything** can be done with function composition?... It's called **lambda calculus**.*

—Joseph McKinsey

The Lambda Calculus

The λ-calculus is a mathematical language of **lambda terms** bound by a set of transformation rules. The λ-calculus notation was introduced in the 1930s by Alonzo Church.

Just like programming languages, the λ-calculus has rules for what is a valid syntax:

Variables: *A variable (such as x) is valid term in the λ-calculus.*

Abstractions: *If t is a term and x is a variable, then the term $\lambda x.t$ is a lambda abstraction.*

Applications: *If t and s are terms, then ts is the application term of t onto s.*

Anonymous Functions

Lambda abstractions can be thought of as anonymous functions in the λ-calculus.

A lambda abstraction which takes an x and returns a t is written as so:

$$\lambda x.t$$

Example

Suppose in mathematics we define a function $f(x) = x + 2$. This could be written as $(\lambda x.x + 2)$ in the λ-calculus[1]. Of course, this function is anonymous and not bound to the name f.

[1] Of course, we haven't said that either $+$ nor 2 is valid in lambda calculus yet. We will get to that...

Functions are First Class

In the λ-calculus, abstractions are not only first class functions, they are our only way of to encode data.

Abstractions are used to encode everything:

- Numbers
- Booleans (true/false)
- Conses
- ...

Currying

Since abstractions in the λ-calculus may only take one argument, currying is typically used to denote functions of multiple arguments. For example, the function $f(x, y) = x$ might be written as:

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

Further, function application is left-associative, so fxy means $(fx)y$.

Free and Bound Variables

The λ operator (which creates lambda abstractions) binds a variable to wherever it occurs in the expression.

- Variables which are bound in an expression are called **bound variables**
- Variables which are not bound in an expression are called **free variables**

Example

With your learning group, identify the free and bound variables in this expression:

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

Transformations

α-conversion: *Allows variables to be renamed to non-colliding names. For example, $\lambda x.x$ is α-equivalent to $\lambda y.y$.*

β-reduction: *Allows functions to be applied. For example, $(\lambda x.\lambda y.x)(\lambda x.x)$ is β-equivalent to $\lambda y.(\lambda x.x)$.*

η-conversion: *Allows functions with the same external properties to be substituted. For example, $(\lambda x.(fx))$ is η-equivalent to f if x is not a free variable in f.*

Examples: Alpha Equivalence

With your learning group, identify if each of the following are a valid α-conversion. Turn in your answers on a sheet of paper with all of your names at the end of class for learning group participation credit for today.

1. $\lambda x.\lambda x.x \rightarrow \lambda y.\lambda y.y$
2. $\lambda x.\lambda x.x \rightarrow \lambda y.\lambda x.x$
3. $\lambda x.\lambda x.x \rightarrow \lambda y.\lambda x.y$
4. $\lambda x.\lambda y.x \rightarrow \lambda y.\lambda y.y$

Examples: Beta Reductions

Fully β-reduce each of the following expressions:

5. $(\lambda x.\lambda y.\lambda f.fxy)(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)(\lambda x.\lambda y.y)$
6. $(\lambda a.\lambda b.a(\lambda b.\lambda f.\lambda x.f(bfx))b)(\lambda f.\lambda x.fx)(\lambda f.\lambda x.f(fx))$

Church Numerals

Since all data in the λ-calculus must be a function, we use a clever convention of functions (called **Church numerals**) to define numbers:

0: $\lambda f.\lambda x.x$
1: $\lambda f.\lambda x.fx$
2: $\lambda f.\lambda x.f(fx)$
3: $\lambda f.\lambda x.f(f(fx))$

... and so on. In fact, the successor to any number n can be written as:

$$\lambda f.\lambda x.f(nfx)$$

Notice this

Defining numbers as functions in this way allows us to apply a Chuch numeral n to a function to get a new function that applies the original function n times.

Shorthand Notations

While it's not a defined part of the λ-calculus, we define common shorthands for some features:

- $0, 1, 2, \ldots$ are shorthand for their corresponding Church numerals
- $\{\text{SUCC}\} = \lambda n.\lambda f.\lambda x.f(nfx)$

Note

The notation "$=$" above is not a part of the λ-calculus. I'm using it for saying "is shorthand for".

Addition and Multiplication

Adding m to n can be thought of as taking the successor to n, m times. Using our shorthand SUCC, this can be written as:

$$\{\text{ADD}\} = \lambda m.\lambda n.(m\,\{\text{SUCC}\}\,n)$$

Similarly, multiplying m by n can be thought of as repeating ADD n, m times and then applying it to 0, this can be written as:

$$\{\text{MULT}\} = \lambda m.\lambda n.(m(\{\text{ADD}\}\,n)0)$$

Boolean Logic

We use the following convention for true and false:

$$\{\text{TRUE}\} = \lambda x.\lambda y.x$$
$$\{\text{FALSE}\} = \lambda x.\lambda y.y \qquad \text{(Church numeral zero)}$$

From here, we can define some common boolean operators:

$$\{\text{AND}\} = \lambda p.\lambda q.pqp$$
$$\{\text{OR}\} = \lambda p.\lambda q.ppq$$
$$\{\text{NOT}\} = \lambda p.p\ \{\text{FALSE}\}\ \{\text{TRUE}\}$$
$$\{\text{IF}\} = \lambda p.\lambda a.\lambda b.pab$$

(returns a if the predicate is TRUE, b otherwise)

Cons Cells

By convention, we will represent a cons cell as a function that applies its argument to the CAR and CDR of the cons cell. This leads to the shorthand:

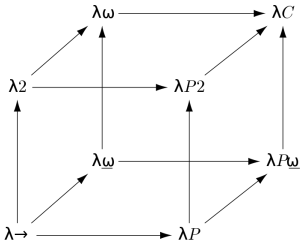
$$\{\text{CONS}\} = \lambda x.\lambda y.\lambda f.fxy$$
$$\{\text{CAR}\} = \lambda c.c\ \{\text{TRUE}\}$$
$$\{\text{CDR}\} = \lambda c.c\ \{\text{FALSE}\}$$
$$\{\text{NIL}\} = \lambda x.\{\text{TRUE}\}$$

Using this, we can define lists:

$$(\{\text{CONS}\}\ 1\ (\{\text{CONS}\}\ 2\ (\{\text{CONS}\}\ 3\ \{\text{NIL}\})))$$

What else is there in Lambda Calculus?

- Getting the predecessor ($\{\texttt{PRD}\}$) for a Church Numeral is hard, but doable (extra credit). To subtract m from n, apply the $\{\texttt{PRD}\}$ function m times to n.
- For recursion, we need to reference ourselves in a lambda abstraction. This is done using a Y-combinator.
- The graduate level Theory of Computation (CSCI 561) class talks much more extensively about the λ-calculus.
- There are many extensions to the λ-calculus such as those encoded by the λ-cube.



Why is any of this Useful?

- λ-calculus can emulate a Turing machine. That means that anything you can do with a classical computer, you can do with the λ-calculus. **This fact underpins all of functional programming.**
- Many functional programming languages (e.g., Haskell, Scheme, SlytherLisp) are just practical implementations of the λ-calculus.
- The λ-calculus gives us another perspective on *type theory* (using the generalization of the λ-calculus called typed λ-calculus).
- It is another way for us to quantify what is computable.