Lambda Calculus

Mattox Beckman

Introduction and Objectives

There are three major paradigms of programming languages that are popular today (and a couple more worth your consideration). Each of these paradigms can be reduced to a specific *model of computation*. In this chapter we will consider the λ-calculus, which is the model of computation for the functional programming languages.

When you are done reading this and working through the exercises, you should be able to explain the three constructs that are part of λ-calculus, show how to simulate arithmetic and recursion, and explain why a computer scientist should be concerned with a theoretical language that can't even be bothered to have integers or booleans.

The Language

Alonzo Church and Stephen Kleene developed λ-calculus in the 1930's to reason about the properties of functions. By giving a fully general set of rules for evaluating a function we are able to reason about the computational aspects of functions. (An alternative is to think of functions as a set of pairs of arguments and values. In this case we emphasize the relation between an argument and a value rather than the computation necessary to convert an argument to a value.)

To this day λ-calculus is very popular with programming language researchers. This is for two reasons. First, λ-calculus is Turing complete—any computation that can be done in a "real" language can be done in λ-calculus. Second, the language is very small, allowing us to focus on the features we want to study without being distracted by the mechanics of the language itself. It is sometimes called "the little white mouse" of programming language research.

The set of lambda terms Λ consists of variables, functions, and function applications. Let M and N represent an arbitrary lambda terms and x represent an arbitrary variable. Then we can define the syntax of Λ as:

$$\begin{array}{rll} \Lambda ::= & x & \text{variables} \\ & MN & \text{function application} \\ & \lambda x.M & \text{functions, a.k.a. } \lambda \text{ abstractions} \end{array}$$

A **variable** in our presentation of λ-calculus will be a symbol of one letter, possibly with "decorations" such as superscripts, subscripts, primes, etc. So, x, y^3, x_n, a', $\hat{z}$ are all examples of valid variable names. $a3$, foo, and bar are not valid, because their names are more than one symbol long.

A **function application** is denoted by juxtaposition. We will usually **not** separate terms with spaces, and instead "run them together." This works because all variable names will be one letter long.

Here are some example function applications: fx, $a(mn)$, fxy The first example is f applied to x. The second example applies m to n, and then applies a to that result. The third example applies f to two variables x and y. Function application associates to the left, so $abcd$ is the same as $((ab)c)d$.

A **function** is denoted by the Greek letter λ (pronounced *LAM-duh*), followed by a parameter name(s), then a period, and then the body of the function. The usual convention is that the body of the function extends as far as it can go. Functions are also called λ-*abstractions* or just *abstractions*.

Here are some example functions:

$\lambda x.x$	The identity function.
$\lambda a.\lambda b.a$	Takes two arguments and returns the first
$\lambda f.\lambda x.f(fx)$	Takes two arguments and applies the first one to the second one twice.

It is common to "stack up" parameters of adjacent λ's. So

$$\lambda a.\lambda b.a \equiv \lambda ab.a$$
$$\lambda f.\lambda x.f(fx) \equiv \lambda fx.f(fx)$$

In longer expressions you may need parentheses to be sure that they are interpreted correctly.

$\lambda x.x\lambda y.y$	This is a function that applies x to the identity.
$(\lambda x.x)\lambda y.y$	This is the identity function applied to the identity function.

Question 1: Which of the following are valid λ expressions?

$\lambda x.xyz$
$\lambda x.\lambda y.$
a
$x\lambda wy.y$
$x\lambda$
$\lambda\lambda xz.zx$
$(abcd)(efgh)ij\lambda klm.mkl$

Free and Bound Variables

When a variable is listed between the λ and the dot, i.e., it is a parameter of a function, then we say that the variable is *bound* to that particular λ. It is possible for there to be more than one λ-binding a variable with the same name. These are counted as two separate variables. [1] In such a case, a particular variable in an expression is bound by the nearest enclosing λ. If a variable is not bound by any λ we say that it is *free*.

[1] Many authors forbid the use of the same variable name in more than one λ to avoid this confusion.

$\lambda x.x\ \lambda y.y$	The x is bound to the first λ.
$(\lambda x.(\lambda x.x)\ x$	The first x is bound to the second λ, and the second x is bound to the first λ.
$\lambda y.\lambda z.x$	The variable x is free.

A free variable will not be given a value by the enclosing expression. You can think of them as being "global."

Question 2: In the following expressions, first use parenthesis to make clear the extent of each λ abstraction, and then indicate to which λ the variable x is bound.

For example, in $\lambda x.\lambda z.x$, we could parenthesize it as $(\lambda x.(\lambda z.x))$, and the x is bound to the first λ.

$\lambda x.\lambda y.x$
$\lambda x.\lambda x.x$
$\lambda x.x\lambda y.x$
$\lambda x.x\lambda x.x$
$\lambda z.x\lambda y.x$
$\lambda z.x\lambda x.x$

Evaluating Expressions using β-reductions

There is only one operation in λ-calculus: β-reduction (sometimes spelled as *beta-reduction*). A β-reduction occurs when you have a λ abstraction applied to another λ term. For example, $(\lambda x.x)y$ is reducible to y. The terms $(\lambda x.xy)$ and $x\lambda y.y$ are not reducible. The first because there is only an abstraction, but nothing after it; the second because the x is not an abstraction.

Be careful when the λ abstraction and the term following it are both arguments to another term. In the term $x(\lambda y.y)z$, the $(\lambda y.y)$ and z are both arguments to x. The $\lambda y.y$ is not being applied to z.

Here's how to perform a β-reduction. Remove the initial λ and its parameter. Then, in the body of that function, replace all occurrences of that variable with the argument. So, any variable that was bound the that lambda gets replaced. Mathematically, we would say

$$(\lambda x.M)N \to [N/x]M$$

Where $[N/x]M$ mean "replace all x's in M that were bound to the $lambda$ by N." It might help to think of using an editor to do a search and replace.

Another notation you might see is this one:

$$(\lambda x.M)N \to M[x := N]$$

It has the same meaning.

Here are a few examples of β reductions. Sometimes one reduction is followed by another one.

$(\lambda x.x)y \to y$
$(\lambda z.x)y \to x$
$(\lambda z.azbz)y \to ayby$
$(\lambda x.(\lambda z.z)x)y \to (\lambda z.z)y \to y$
$(\lambda x.x(\lambda z.ax)(\lambda x.bx))y \to y(\lambda z.ay)(\lambda x.bx)$
$(\lambda x.(\lambda z.zx)(\lambda x.bx))y \to (\lambda z.zy)(\lambda x.bx) \to (\lambda x.bx)y \to by$

Question 3: Try doing these reductions. Reduce each expression as much as you can.

$(\lambda x.xx)y$
$(\lambda x.axxa)y$
$(\lambda x.(\lambda z.zx)q)y$
$(\lambda x.x((\lambda z.zx)(\lambda x.bx)))y$
$(\lambda a.a)(\lambda b.b)(\lambda c.cc)(\lambda d.d)$

A note about reduction order You might be wondering what to do if you have a choice about which application to make. Consider this example:

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

We will be using *normal order reduction*, which says that the leftmost, outermost reduction is performed first. In this case, the λx is applied first.

Another option is *applicative order*. In that system, the leftmost, innermost reduction is performed first. This would be the λa reduction. Essentially, this is "call by value," as the arguments to a function are evaluated before the function is called.

For now, we are not going to emphasize the different reduction orders. Use normal order reduction for everything, i.e., don't reduce the arguments before calling the function.

Finally, consider this example:

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

Do we do the λy reduction or not? In other words, do we do computation inside of a function before the function has been called? Most languages say "no."[2] If we say "yes," then we get $\lambda x.xx$, and say that the result is in *normal form*. This is the style we will use.

[2] This is known as *weak head normal form*, but we are not going to talk about that in this course.

Alpha Capture and Renaming

Consider the following two λ terms:

$$X \equiv (\lambda a.\lambda b.ab) \qquad \text{and} Y \equiv (\lambda w.\lambda x.wx)$$

Clearly (we hope!) terms X and Y do not differ at all in meaning. The names of the variables are different, but what these functions actually *do* when you apply them is identical. When two terms have the same structure and differ only in the names of the variables, they are said to be α*-equivalent*. If we rename the variables in a term in such a way that the structure is preserved, is is called α*-renaming*.

Here is an example of two terms that are **not** α-equivalent.

$$X \equiv (\lambda a.\lambda b.ab) \qquad \text{and} Y \equiv (\lambda w.\lambda x.xw)$$

Notice how the second term Y applies its arguments in the reverse order compared to X. The structure of the terms are different, so they are not α-equivalent.

Question 4: Which of the following pairs are α-equivalent?

$$\begin{array}{ll} \lambda a.\lambda b.abb & \lambda b.\lambda a.baa \\ \lambda a.\lambda b.abb & \lambda i.\lambda j.jji \\ \lambda x.x\lambda y.x & \lambda e.e\lambda f.f \\ \lambda x.x\lambda y.x & \lambda e.e\lambda f.e \end{array}$$

Sometimes it happens that a free variable or a variable that is bound to one λ ends up being moved around the expression in such a way that it gets "captured" by another λ. This is known as α*-capture*, and is almost always a bad thing. Here are some examples.

$$(\lambda x.\lambda y.yx)y \to \lambda y.yy$$

The free variable y has been captured by the λy. Contrast this example, where we have α-renamed the λy term to λz:

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

In general, if you are performing a reduction and find that you are going to capture a free variable, you need to α-rename the capturing lambda. You cannot rename the free variable!

You can see that the non-capturing version has a different structure than the capturing version.

It is easy to write examples that cause capture when you use free variables, but you can do it with only bound variables too, if you reduce to normal form.

$$(\lambda f.\lambda x.fx)(\lambda y.\lambda x.y) \to \lambda x.(\lambda y.\lambda x.y)x \to \lambda x.\lambda x.x$$

We can prevent this by α-renaming the second term.

$$(\lambda f.\lambda x.fx)(\lambda y.\lambda z.y) \to \lambda x.(\lambda y.\lambda z.y)x \to \lambda x.\lambda z.x$$

For this reason, it is best that you always use distinct variable names for your lambdas.

Church Numerals

The λ-calculus doesn't have numbers, but we can model them using functions. We need to know what a number actually is in order to model it. For our model, we will say that a number n can be thought of as a potential: someday we are going to do something n times. If we are going to do something, we also need something to do it to.

Therefore, a *numeral*[3] in λ calculus will be a function that takes two arguments, an action and a target, and performs the action a certain number of times to the target. These are called Church Numerals, after Alonso Church.

f0 = $\lambda f\ x\ .\ x$

f1 = $\lambda f\ x\ .\ (f\ x)$

f2 = $\lambda f\ x\ .\ (f\ (f\ x))$

f3 = $\lambda f\ x\ .\ (f\ (f\ (f\ x)))$

When working with these in Haskell, it is nice to have a means of viewing them.

```
c0 f x = x
c1 f x = f x
c2 f x = f (f x)
show n = n (+ 1) 0
```

```
Prelude> show c2
2
```

[3] A number is the abstract concept of quantity. A numeral is a textual or other representation of a specific number. When you see the character 5 you are looking at a numeral, but you think of the number it represents.

Incrementing Church Numerals

To increment a Church Numeral, we need to take the church numeral m, the intended action f, and the intended target x. Think for a moment how can we cause f to be applied to x a total of $m + 1$ times?

We can cause m applications by calling $(m\ f\ x)$. We can apply f to the result for a total of $m + 1$ calls.

```
inc m f x = f (m f x)
```

Adding Church Numerals

Similar reasoning can yield addition and multiplication. Try to work out addition first before looking at the answer. You will take two church numerals, m and n, as well as the action f and the target x.

```
cAdd m n f x = m f (n f x)
```

Multiplication is simply repeated addition. To do multiplication, consider what the fuction $(m\ f)$ represents. It itself is a function that will perform f a total of m times. We can repeat this function as well as we could repeat any other. Try it.

```
cMul m n f x = m (n f) x
```

Subtraction is *much* more complex.

To Infinity and Beyond

Suppose we want to implement

```
f n = f (n+1)
```

The outline of the function would look like

$$\lambda n.(f\ (inc\, n))$$

But, how does f get to know itself?
The only way is to tell f its own name by having it take it as a parameter.

$$\lambda f.\lambda n.(f\ (inc\, n))$$

Once f is written this way, we finish by passing a copy of f to itself.

$$\begin{array}{l}(\lambda f.\lambda n.(f\ (inc\, n)))\\ \lambda f.\lambda n.(f\ (inc\, n))\end{array}$$

But now f must pass itself into itself... so we have

$$\begin{array}{l}(\lambda f.\lambda n.((f\ f)\ (inc\, n)))\\ \lambda f.\lambda n.((f\ f)\ (inc\, n))\end{array}$$

You may find it useful to trace out the call $f\ c_2$ to be sure you understand what's happening.

The Y-combinator

It kind of painful to have to double the size of each function to enable recursion. Perhaps we can use another function to do that for us.

We'll call this function Y. It will take another function f and return a recursive version of it. Function f will still need to ask for itself as a parameter, but Y can be responsible for duplicating it.

We want it to be a *combinator*, a function that produces its result only through function application.

Therefore, Y should have the following property.

$$(Y\ f)\ \rightarrow\ (f\ (Y\ f))$$

Using the doubling trick from above, we implement Y like this:

$$Y\ =\ \lambda f.(\lambda y.(f\ (y\ y))\ \lambda y.(f\ (y\ y)))$$

Tracing out $(Y\ F)$, we get

$$\begin{aligned}(Y\ F) &= (\lambda f.(\lambda y.(f\ (y\ y))\ \lambda y.(f\ (y\ y)))\ F)\\ &= (\lambda y.(F\ (y\ y))\ \lambda y.(F\ (y\ y)))\\ &= (F\ (\lambda y.(F\ (y\ y))\lambda y.F\ (y\ y)))\\ &= (F\ (Y\ F))\end{aligned}$$

which is what we wanted.

Lambda Calculus in Real Programming Languages

You have λ expressions in "normal" programming languages too; they are even being retrofitted onto languages that did not have them previously, such as Java and C++. Here are some examples of where you will see them. Each one will be the "increment" function anonymously applied to 10 to get 11.

Haskell `(\x -> x + 1) 10`

OCaml `(fun x -> x + 1) 10`

Common Lisp `(funcall (lambda (x) (+ x 1)) 10)`

Javascript `(function(x){return x+1;})(10)`

Clojure Variation 1

```
(#(+ 1 %) 10)
```

Clojure Variation 2

```
((fn [x] (+ 1 x)) 10)
```

Python `(lambda x: x + 1) (10)`

Ruby `lambda { |x| x + 1}.call (10)`

C++ 11 `#include <iostream>`

```
using namespace std;

int main() {
   cout << [](int x){ return x + 1; }(10) << endl;
}
```

Java `(x) -> { x + 1};`

There are other things you have to do to actually *call* this in Java, though.

Exercises

Solutions to exercises

Solution 1

Valid	$\lambda x.xyz$
Not Valid	$\lambda x.\lambda y.$
Valid	a
Valid	$x\lambda wy.y$
Not Valid	$x\lambda$
Not Valid	$\lambda\lambda xz.zx$
Valid	$(abcd)(efgh)ij\lambda klm.mkl$

Solution 2

$(\lambda x.(\lambda y.x))$	The first λ.
$(\lambda x.(\lambda x.x))$	The second λ.
$(\lambda x.x(\lambda y.x))$	Both are bound to the first λ.
$(\lambda x.x(\lambda x.x))$	The first x is bound to the first λ, the second x is bound to the second λ.
$(\lambda z.x(\lambda y.x))$	Both x's are free.
$(\lambda z.x(\lambda x.x))$	The first x is free but the second is bound to the second λ.

Solution 3

$(\lambda x.xx)y \rightarrow yy$
$(\lambda x.axxa)y \rightarrow ayya$
$(\lambda x.(\lambda z.zx)q)y \rightarrow (\lambda z.zy)q \rightarrow qy$
$(\lambda x.x((\lambda z.zx)(\lambda x.bx)))y \rightarrow y((\lambda z.zy)(\lambda x.bx)) \rightarrow y((\lambda x.bx)y) \rightarrow y(by)$
$(\lambda a.a)(\lambda b.b)(\lambda c.cc)(\lambda d.d) \rightarrow (\lambda b.b)(\lambda c.cc)(\lambda d.d) \rightarrow (\lambda c.cc)(\lambda d.d) \rightarrow (\lambda d.d)(\lambda d.d) \rightarrow (\lambda d.d)$

Solution 4

Yes	$\lambda a.\lambda b.abb$	$\lambda b.\lambda a.baa$
No	$\lambda a.\lambda b.abb$	$\lambda i.\lambda j.jji$
No	$\lambda x.x\lambda y.x$	$\lambda e.e\lambda f.f$
Yes	$\lambda x.x\lambda y.x$	$\lambda e.e\lambda f.e$

Colophon

This document was compiled using Lua LaTeX and the `tufte-book` package. The body text is set in the *Equity* font, and the headers are set in the *Concourse* font. Both these fonts are available from Matthew Butterick. The source code is set in *Computer Modern Teletype*, designed by Donald Knuth.