Greek Literature:
Functional Fundamentals from Lambda Calculus

Adam McCullough

@TheWizardTower ... most places

Front matter

This is meant to be for novices.

No experience assumed.

Objective

Most guides cover the how -- poorly

Objective

Most guides cover the how -- poorly

I want to cover the **why**, as well as the how

Step One
Identity

$\lambda x.x$

Step One
Identity

λx.x

λ - Begin a “Lambda Abstraction”

$\lambda \textcolor{red}{\mathbf{x}}.x$

x - Take one argument, bound to the name ‘x’

$\lambda x.x$

. - Separate the function ‘head’ from the function ‘body’

$$\text{Head} \rightarrow \lambda x.x \leftarrow \text{Body}$$

. - Separate the function ‘head’ from the function ‘body’

$\lambda x.\mathbf{x}$

X - Function body. What to do with the argument.

$\lambda x.\mathbf{x}$

X - Function body. What to do with the argument.

In this case, nothing.

Putting that all together...

Putting that all together...

This is a function (“Lambda Abstraction”) that:

Putting that all together...

This is a function ("Lambda Abstraction") that:

- Takes one argument

Putting that all together...

This is a function (“Lambda Abstraction”) that:

- Takes one argument
- Binds the value of that argument to ‘x’,

Putting that all together...

This is a function (“Lambda Abstraction”) that:

- Takes one argument
- Binds the value of that argument to ‘x’,
- Then immediately returns it.

```
\x -> x
```

Pleasantly enough, it’s very similar in Haskell

```
\x -> x
```

- The ‘\’ is shorthand for λ

```
\x -> x
```

- The ‘x’ means the same thing:
  - It takes an argument
  - Binds it to the name ‘x’

\x -> x

- Equivalent to ‘.’

```
\x -> x
```

- Function body.

Algebra!

Algebra!

$$f(x) = x$$

Question!

$\lambda x.x$

Vs

$\lambda y.y$

Is there a difference?

Turns out, no

This is called “Alpha Equivalence”

Well that’s cool…

Well that's cool…

..but how does it work?

Well that's cool…

..but how does it work?

Let's run through an example.

$(\lambda x.x)2$

$(\lambda x.x)2$

(the parens appeared to disambiguate between function and argument)

$(\lambda x.x)2$

This is a valid Lambda Calculus Expression…

$(\lambda x.x)2$

This is a valid Lambda Calculus Expression…

But we can simplify it, by doing Beta Reduction.

Beta reduction

Like much of Algebra, it’s mechanical.

Beta reduction

1. Bind the argument to the name

Beta reduction

1. Bind the argument to the name
2. Strip the function head and separator

Beta reduction

1. Bind the argument to the name
2. Strip the function head and separator
3. Repeat as arguments and lambdas meet

$(\lambda x.x)2$

$(\lambda x.x)2$

$\lambda 2.2$

$$(\lambda x.x)2$$

Head $\rightarrow$ $\lambda 2.2$ $\leftarrow$ Body

$(\lambda x.x)2$

$\lambda 2.2$

2

What happens if we pass the identity function to itself?

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

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

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

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

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

$$(\lambda y.y)$$

Next step:

Functions with Multiple Arguments

It’s tempting to just do the “obvious” thing:

λxy.x y

But alas, not so much.

Lambda Abstractions take one argument

(So does every function in Haskell)

This is because functions can return functions

So, a function with two arguments is:

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

Let’s step through that

```
(λx.
   (λy.
      x y))
  (λz.z) 2
```

$$
\begin{aligned}
&(\lambda \mathbf{x}. \\
&\quad (\lambda y. \\
&\qquad \mathbf{x}\ y)) \\
&\quad \mathbf{(\lambda z.z)}\ 2
\end{aligned}
$$

$$
\begin{aligned}
&(\lambda\,\color{blue}{\boldsymbol{(\lambda z.z)}}.\\
&\quad(\lambda y.\\
&\qquad\color{blue}{\boldsymbol{(\lambda z.z)}}\ y))\\
&\quad 2
\end{aligned}
$$

$$
\begin{aligned}
&(\lambda y. \\
&\quad (\lambda z.z)\ y)) \\
&2
\end{aligned}
$$

$(\lambda y.$

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

2

$$(\lambda 2.$$
$$\quad (\lambda z.z)\ 2)$$

$$(\lambda z.z)\ 2$$

$$(\lambda z.z)\ 2$$

λ2.2

2

Converting a function with multiple arguments into this form is called "Currying"

This can even be done with traditional mathematical functions!

Partial application [edit]

Currying resembles the process of evaluating a function of multiple variables, when done by hand, on paper, being careful to show all of the steps.

For example, given the function $f(x, y) = y/x$:

> To evaluate $f(2, 3)$, first replace x with 2
>
> Since the result is a function of y, this new function $g(y)$ can be defined as $g(y) = f(2, y) = y/2$
>
> Next, replace the y argument with 3, producing $g(3) = f(2, 3) = 3/2$

On paper, using classical notation, this is usually done all in one step. However, each argument can be replaced sequentially as well. Each replacement results in a function taking exactly one argument.

This example is somewhat flawed, in that currying, while similar to partial function application, is not the same (see below).

https://en.wikipedia.org/wiki/Currying

$$f(x,y) = y / x$$

$$f(x,y) = y / x$$

$$g(y) = f(2,y) = y / 2$$

$$f(x,y) = y / x$$

$$g(y) = f(2,y) = y / 2$$

$$g(3) = f(2,3) = 3 / 2$$

What if we only supplied one argument?

```
(λx.
   (λy.
      x y))
  (λz.z)
```

$$(\lambda \mathbf{x}.\ (\lambda y.\ \mathbf{x}\ y))\ (\boldsymbol{\lambda} \mathbf{z}.\mathbf{z})$$

$$
\begin{aligned}
&(\lambda\,\color{blue}{\boldsymbol{(\lambda z.z)}}.\\
&\quad(\lambda y.\\
&\qquad\color{blue}{\boldsymbol{(\lambda z.z)}}\ y))
\end{aligned}
$$

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

Haskell Example!

```
let myConst =
    (\a -> (\b -> a))
```

We can implement this in Lambda Calculus!

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

(Another) Haskell Detour

Let's talk about Types

ghci commands

:t <value>

“What is the type of this value?”

```
Prelude> :t const
const :: a -> b -> a
```

```
const :: a -> b -> a
```

`const` is the name of the value

```
const :: a -> b -> a
```

“ : : ” is read as “has type”

```
const :: a -> b -> a
```

a is the type identifier of the first arg

```
const :: a -> b -> a
```

`->` means the same as in a value

```
const :: a -> b -> a
```

b is the type name of the next arg.

```
const :: a -> b -> a
```

```
The type of a == a!
```

ghci commands

:t <value>

"What is the type of this value?"

ghci commands

:k <type>

“What is the kind of this type?”

```
Prelude> :k Integer
Integer :: *
```

```
data Pair a b = Pair a b
```

```
data Pair a b = Pair a b
```

```
Prelude> :k Pair
Pair :: * -> * -> *
```

```
Prelude> :k Pair Integer
Pair :: * -> *
```

```
Prelude> :k Pair Integer String
Pair :: *
```

Next Steps:
Recursion

(well, almost)

No:

- If

No:

- If
- Loops (For, While, ForEach, etc)

No:

- If
- Loops (For, While, ForEach, etc)
- Recursion

No:

- If
- Loops (For, While, ForEach, etc)
- Recursion
- Booleans

No:

- If
- Loops (For, While, ForEach, etc)
- Recursion
- Booleans
- Strings

No:

- If
- Loops (For, While, ForEach, etc)
- Recursion
- Booleans
- Strings
- Numbers (!!)

Lambda Calculus is a Turing Tarpit

You can represent Booleans and Numbers with some cleverness

But it isn’t built in.

Let’s talk about the Omega combinator

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

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

<Beta reduction>

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

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

<Beta reduction>

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

<Alpha Conversion>

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

This is close, but not quite what we want

This is close, but not quite what we want

What we need is the Y combinator

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

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

Look familiar?

$$(\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ \text{factorial}$$

$$
\begin{aligned}
&(\lambda f. \\
&\qquad (\lambda x.\ f\ (x\ x)) \\
&\qquad (\lambda x.\ f\ (x\ x)))\ \textbf{factorial}
\end{aligned}
$$

```
(λfactorial.
    (λx. factorial (x x))
    (λx. factorial (x x)))
```

(λx. **factorial** (x x))
(λx. **factorial** (x x))

(λx. factorial (x x))
(λx. factorial (x x))

$(\lambda \mathbf{x}.\ \mathrm{factorial}\ (\mathbf{x}\ \mathbf{x}))$

$\boldsymbol{(\lambda x.\ \mathbf{factorial}\ (x\ x))}$

(λ**(λx. factorial (x x))**.
factorial
(λx. factorial (x x))
(λx. factorial (x x)))

factorial

(λx. factorial (x x))

(λx. factorial (x x))

factorial

$(\lambda x.\ \text{factorial}\ (x\ x))$

$(\lambda x.\ \text{factorial}\ (x\ x))$

Thank you.

Also, we’re hiring!