Numbers in Lambda Calculus

Ken Q Pu, Faculty of Science, Ontario Tech University

Numbers

Encoding of numbers with pebbles

Arithmetic

Additions can be done by physically piling up the pebbles together.

Numbers

But pebbles cannot do other arithmetic operations very well

×

= ?

What is the physical process to compute multiplication?

÷

= ?

Numbers

Encoding of numbers with pebbles

Arithmetics

+

=

Numbers

Binary Encoding

0001 0010 0011 0100

*These are **string**.* So we can write them down on the tape of a Turing machine.

Arithmetics

$0010 \times 0011 = ?$

We can use a multiplication TM.
The result is the tape content after it halts.

Numbers: Lambda Calculus Encoding

Remember, everything is a function with a single input.

Numbers: Lambda Calculus Encoding

\f.\x. x

$x \longrightarrow x$

Nothing is applied to the input.

\f.\x. f x

$x \longrightarrow \boxed{f} \longrightarrow f(x)$

*Apply **f** once to the input **x**.*

\f.\x. f (f x)

$x \longrightarrow \boxed{f} \longrightarrow \boxed{f} \longrightarrow f(f(x))$

*Apply **f** twice to the input **x**.*

\f.\x. f (f (f x))

$x \longrightarrow \boxed{f} \longrightarrow \boxed{f} \longrightarrow \boxed{f} \longrightarrow f(f(f(x)))$

*Apply **f** three times to the input **x**.*

Ken Q Pu, Faculty of Science, Ontario Tech University

Numbers: Church Encoding

\f.\x. x — 0

\f.\x. f x — 1

\f.\x. f (f x) — 2

\f.\x. f (f (f x)) — 3

...

We need to define addition in lambda calculus

\f.\x. f x	+	\f.\x. f (f x)

→ | \f.\x. f (f (f x)) |

such that its normal form is the encoding of the sum

Ken Q Pu, Faculty of Science, Ontario Tech University

Functional Semantics of Church Numbers

3 is \f.\x. f (f (f x))

3 is a function with two inputs*. It applies the first argument to the second argument three times.

All numbers are functions with two arguments. The first argument is repeatedly applied to the second argument *n* times.

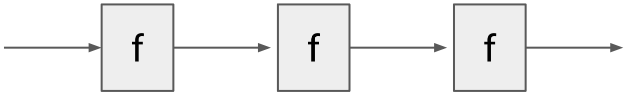
Arithmetics of Church Numbers

Successor function

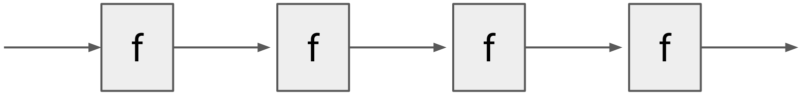
In mathematics:

Succ(n) = n + 1

In Lambda Calculus:



\f.\x. f (f (f x))



\f.\x. f (f (f (f x)))

Succ = \n. (n+1)
= \n. \f.\x. f (n f x)

Keep in mind that n and (n+1) are functions with two inputs.

Understanding the Successor function

```
Succ = \n.\f.\x. f (n f x)
```

*The first parameter is a Church number. So, **n** is a curried function with two inputs.*

*The body evaluates to a Church number, which means that it is a curried function with two parameters: **f** and **x**.*

Ken Q Pu, Faculty of Science, Ontario Tech University

Understanding the Successor function

```
Succ = \n.\f.\x. f (n f x)
```

So **(f (n f x))** *applies*
f *to* **x** (n+1)-*times.*

Since **n** *is a Church number, it repeatedly applies* **f** *to* **x** *n-times.*

Lambda Calculus has no variables

```
Succ = \n.\f.\x. f (n f x)
```

Pure LC does not permit variables.
So, here ***Succ*** *is just a short-hand for*
the expression **\n.\f.\x. f (n f x)***.*

```
(Succ (\f.\x. (f x)))

((\n.\f.\x. f (n f x)) (\f.\x. (f x)))
```

How far can we push computation in LC?

We can now compute the **next** Church number from an input Church number using the Succ expression.

What about general programming?

Multiplication	Boolean values: True and False	If-else
Exponentiation	Propositional logic: AND, OR, NOT	For-loop
	Greater than, less than, Equal	While-loop