Lecture 28: Church Numerals

Related Data Structures

MATH230

Te Kura Pāngarau | School of Mathematics and Statistics
Te Whare Wānanga o Waitaha | University of Canterbury

Outline

Church Numerals

All that is available to us is application and abstraction. However, that is enough to encode the behaviour of the natural numbers in the λ-calculus.

$$
\begin{aligned}
\mathsf{ZERO} &= \lambda s.\ \lambda x.\ x \\
\mathsf{ONE} &= \lambda s.\ \lambda x.\ s(x) \\
\mathsf{TWO} &= \lambda s.\ \lambda x.\ s(s(x)) \\
\mathsf{THREE} &= \lambda s.\ \lambda x.\ s(s(s(x))) \\
&\vdots \\
n &= \lambda s.\ \lambda x.\ s(s \ldots (s(x)) \ldots) \\
&\vdots
\end{aligned}
$$

The Church numeral representing the natural number M is a binary λ-expression that applies the first argument to the second M times.

Pattern

If n is a Church numeral and f, a are arbitrary function symbols, then we have the following patterns that occur often in computations with Church numerals.

The λ-expression nf is a function which applies f n times to its input. That is, f composed with itself n times.

So we can read

$$nfa$$

as "f" *applied to* "a" *repeatedly* "n" *times*.

If we follow this idea when f is also a Church numeral, then we get a λ-expression for exponentiation.

Observation

$$
\begin{aligned}
\text{TWO ONE} &= (\lambda u.\ \lambda v.\ u(u(v)))(\lambda s.\ \lambda x.\ s(x)) \\
&= \lambda v.\ (\lambda s.\ \lambda x.\ s(x))((\lambda s.\ \lambda x.\ s(x))(v))) \\
&= \lambda v.\ (\lambda s.\ \lambda x.\ s(x))(\lambda x.\ v(x)) \\
&= \lambda v.\ (\lambda s.\ \lambda x.\ s(x))(\lambda w.\ v(w)) \\
&= \lambda v.\ (\lambda x.\ (\lambda w.\ v(w))(x)) \\
&= \lambda v.\ \lambda x.\ v(x) \\
&= \text{ONE}
\end{aligned}
$$

$$
\begin{aligned}
\text{ONE TWO} &= (\lambda s.\ \lambda x.\ s(x))(\lambda u.\ \lambda v.\ u(u(v))) \\
&= (\lambda x.\ (\lambda u.\ \lambda v.\ u(u(v)))(x)) \\
&= (\lambda x.\ \lambda v.\ x(x(v)))) \\
&= \text{TWO}
\end{aligned}
$$

Example: Exponential

We abstract over this idea to define a λ-expression to compute exponentiation.

$$\text{EXP} = \lambda e.\ \lambda b.\ eb$$

Example

β-reduce the following expression to normal form

EXP THREE TWO

Programming in λ-Calculus

When programming and running computations in this language we do not update named spaces in memory.

We can't think about updating a number stored in a named variable. There is no syntax for this updating in the λ-calculus.

Each time we calculate a new λ-expression (e.g. Church numeral) we must construct it, from scratch, using the input numerals.

Encoding Arithmetic Functions

We will now find λ-expressions for fundamental arithmetic functions and predicates on Church numerals.

Arithmetic Functions

SUCC

SUM

MULT

EXP

PRED

SUB

Arithmetic Predicates

ZERO?

GREATER?

EQUAL?

Encoding Arithmetic Functions

Programs in the λ-calculus need to **construct** the output.

Unary functions on Church numerals will always start

$$\lambda n.\ \lambda u.\ \lambda v.\ \langle \mathsf{BODY} \rangle$$

Binary functions on Church numerals will always start

$$\lambda m.\ \lambda n.\ \lambda u.\ \lambda v.\ \langle \mathsf{BODY} \rangle$$

The first abstractions are for the inputs to the function.

Second abstractions (u,v) are to construct the output numeral.

Successor

The successor is a unary function that returns a numeral with one more function application of the first argument to the second.

$\text{SUCC} = \lambda n.\ \lambda u.\ \lambda v.$

Example

SUCC ZERO

SUM

The sum of two Church numerals m, n is a binary function that returns a numeral with $m + n$ applications of the first argument to the second. This is similar to string concatenation of successors.

$\mathsf{SUM} = \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.$

Example

SUM ONE ONE

MULT

If m, n are Church numerals, then the output of multiplication requires n applications m times of the first argument to the second.

$\text{MULT} = \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.$

Example

MULT TWO TWO

ZERO?

Given Church numeral m how do we test if it is ZERO?

$\text{ZERO?} = \lambda m.$

Predecessor

To one way of thinking, we need to *remove* one application of the function in the Church numeral.

However that way of thinking is “state based” - as if we have an object somewhere in some memory and we update its properties.

This is not the way programming is done in the λ-calculus.

Instead we need to think, given an input Church numeral n how do we construct the Church numeral representing $n-1$?

PAIR

We have been treating applications of the form ab as if they were pairs. Let us formalise this idea with a function to CONStruct a pair from two inputs.

$$\text{PAIR} = \lambda x.\lambda y.\lambda f.\ f\ x\ y$$

Once a pair is constructed, we may use the following methods to retrieve either the first or second element respectively.

$$\text{FST} = \lambda u.\ \lambda v.\ u \qquad \text{SCD} = \lambda u.\ \lambda v.\ v$$

Example
PAIR ONE TWO FST $=_\beta$ ONE
PAIR ONE TWO SCD $=_\beta$ TWO
PAIR ONE (PAIR TWO THREE) SCD $=_\beta$ PAIR TWO THREE

PRED

We now have the data structure required to implement the algorithm for calculating the predecessor of a Church numeral.

First we write a function which takes in a pair $p = (a, b)$ of Church numerals and outputs the pair consisting of the successor of the first (SUCC a) in the pair, together with the first a in the pair.

$$\Psi = \lambda p.\ \text{PAIR (SUCC } (p \text{ FST})) \ (p \text{ FST})$$

Now we need to iterate this n times on the input pair ZERO ZERO and retrieve the second element.

$$\text{PRED} = \lambda n.\ (n\ \Psi\ \text{(PAIR ZERO ZERO)})\ \text{SCD}$$

Example

PRED ONE

SUB

Given Church numerals m, n how do we construct the Church numeral representing $m - n$?

$\mathsf{SUB} = \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.$

Example

SUB TWO ONE

GREATER?

Given Church numerals m, n how do we test if one is larger than the other?

GREATER? $= \lambda m.\ \lambda n.$

Example

GREATER? ONE ONE

EQUAL?

Given Church numerals m, n how do we test if they are equal?

EQUAL? $= \lambda m.\ \lambda n.$

Example

EQUAL? ONE ZERO

Summary

Arithmetic Functions

$$
\begin{aligned}
\text{SUCC} &= \lambda n.\ \lambda u.\ \lambda v.\ u(nuv) \\
\text{SUM} &= \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.\ mu(nuv) \\
\text{MULT} &= \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.\ m(nu)v \\
\text{EXP} &= \lambda e.\ \lambda b.eb \\
\text{PRED} &= \lambda n.\ (n\ \Psi\ (\text{PAIR ZERO ZERO}))\ \text{SCD} \\
\text{SUB} &= \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.\ (n\ \text{PRED}\ m)\ u\ v
\end{aligned}
$$

Arithmetic Predicates

$$
\begin{aligned}
\text{ZERO?} &= \lambda m.\ m\ (\lambda x.\ \text{FALSE})\ \text{TRUE} \\
\text{GREATER?} &= \ \lambda m.\ \lambda n.\ \text{ZERO?}\ (\text{SUB}\ n\ m) \\
\text{LESS?} &= \ \lambda m.\ \lambda n.\ \text{ZERO?}\ (\text{SUB}\ m\ n) \\
\text{EQUAL?} &= \lambda m.\ \lambda n.\ \text{AND}\ (\text{GREATER?}\ n\ m)\ (\text{LESS?}\ n\ m)
\end{aligned}
$$

Further Reading

Here are some recommended reading to follow up on the lecture content. Some are freely available online.

Type Theory and Functional Programming, *Simon Thompson*

- Stanford Encyclopedia Articles:
  1. The Lambda Calculus
  2. Type Theory
  3. Church's Type Theory

Church's original papers are worth visiting, although more work than Turing's paper.
An Unsolvable Problem of Elementary Number Theory, *Alonso Church*

A Note on the Entscheidungsproblem, *Alonso Church*