λ Calculus

MATH230

Te Kura Pāngarau
Te Whare Wānanga o Waitaha

Outline

What is computation?

Gödel Encoding

Since data can be encoded to natural numbers, questions about the computability of proofs come down to questions about the computablility of functions on the natural numbers.

Language for Computation

What was wanted by the mathematicians of the early 1900s was a single precise language which could express *all possible computations* on the natural numbers.

Alan Turing	Turing Machine
Stephen Kleene	μ Recursive Functions
Alonso Church	λ-Calculus

It was quickly shown that any function expressible in one of these languages was also expressible in each of the others. In other words, they all agree on which procedures are “computable”.

Church-Turing-Kleene Thesis: a function on the natural numbers can be calculated by an “effective procedure” if and only if it is computable by a Turing machine.

Church's Thesis

Since then many languages for computation have been defined.

- Register machines
- Finite state automata
- Post machines
- Python
- C
- Java
- FRACTRAN

None have been shown more expressive than Turing machines.

Lambda Calculus

In this course we will stick to studying lambda calculi.

Other models, like Turing machines, are very fun to work with! However, other courses at UC already teach you about these and other models. Lambda calculi are not taught elsewhere.

Moreover, focusing on the lambda calculi leads us to interesting ideas about the possibility of proof and the future of mathematics.

What is Computation?

Recall the natural deductions we did from the axioms of Peano Arithmetic. Many of the steps we performed in those proofs were of one of the following forms:

- $\forall$ E - substituting terms.
- $=$ E - substituting identical terms.
- $\forall$ I - abstracting over patterns.

The process of computing proofs of those theorems largely consisted of substitution steps. Computation was substitution.

If you reflect on the many computations that you've done over your lifetime, you will find that this is a large part of what you do too.

Alonso Church's λ-calculus can be thought of as a formalisation of this observation.

Lambda Calculus

The lambda calculus is a formal language with rules of formation, manipulation, and simplification of strings called λ-expressions. All programs we write, and computations we carry out, will be expressed in this language.

It is the process of "simplification" - known as β-reduction - that is the process of computation in this model.

Some of the ideas of the lambda calculus go back to Gottlob Frege, but Alonso Church took those ideas and developed the theory proper through a series of papers in the 1930s.

This (with its typed counterparts) has fascinating links to proofs in first-order logic and through this to program and proof verification.

Intuition

Everything is a function.

Mathematics	λ-Calculus
$f(x) = x^2 + 3$	$\lambda\ x.\ x^2 + 3$
$g(x, y) = x^2y + 2y$	$\lambda\ x.\ (\lambda\ y.\ x^2y + 2y)$
$f(3)$	$(\lambda\ x.\ x^2 + 3)\ (3)$
$g(4, 2)$	$((\lambda\ x.\ (\lambda\ y.\ x^2y + 2y))\ (4))\ (2)$
0	$\lambda\ s.\ (\lambda x.\ x)$
3	$\lambda\ s.\ (\lambda x.\ s\ (s\ (s\ x)))$

Grammar of Lambda Calculus

Lambda calculus has countably many variables $x, y, z, a, b, c, \ldots$

Terms of the λ-calculus i.e. λ-terms are constructed using the following three mechanisms.

Variables: $x, y, z, \ldots$

Application: $(t\ u)$ for λ-expressions t, u

Abstraction: $(\lambda x.\ e)$ for variable x and λ-expression e

The term e in the λ abstraction is referred to as the body of the abstraction.

Examples

g

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

$((\lambda x.\ x\ x)\ g)$

BNF λ-Terms

Syntax Trees

λ-terms can be represented in a syntax tree.

Notation Conventions

As with well-formed formulae in logic, we may opt to drop brackets. All λ-expressions will be interpreted under the following conventions.

Application associates to the left e.g.

$$t\ u\ v = ((t\ u)\ v)$$

Abstraction associates to the right e.g.

$$\lambda x.\lambda y.\lambda z.\ t \ = \ \lambda x.(\lambda y.(\lambda z.\ t))$$

Application binds tighter than abstraction e.g.

$$\lambda x.\ t\ u \ = \ \lambda x.\ (t\ u) \ \neq \ (\lambda x.\ t)u$$

Keep all brackets if that helps.

Interpreting λ-expressions

All λ-expressions are to be thought of as functions.

Application, $(t\ u)$, is thought of as applying t to the input u.

Abstraction $(\lambda x.\ u)$ is a function which takes input into x and substitutes that value in every (free) instance of x in the λ-expression u.

This means we have analogous free, bound, and substitution definitions as discussed for first-order logic and quantifiers.

Free & Bound Variables

The λ abstraction operator binds instances of its variable within the body of the abstraction expression. If x is a variable and e is a λ-expression, then x is bound in any expression of the form

$$\lambda x.\ e$$

Any variable in e that is not bound by an abstraction is called free. Instances of a variable bound by an abstraction are said to be in the *scope* of the abstraction.

Examples

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

$$(\lambda z.\ z\ x)(x)$$

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

Substitution

$M = (x\ x) \qquad N = (\lambda x.\ x + z)(\lambda x.\ x)(\lambda y.\ y\ x) \qquad t = \lambda z.\ z$

Substitutions of a term t occur for all free instances of a variable in another expression M.

$M[x := t]$

$N[x := t]$

Substitution Rules

When we substitute one λ-expression N into another λ-expression M for a variable x we replace all *free occurences* of x in M with N: this is denoted $M[x := N]$.

Substitution of λ-expressions is defined recursively as follows

$$x[x := N] = N$$

$$y[x := N] = y \text{ when } y \neq x$$

$$(M_1\ M_2)[x := N] = (M_1[x := N]\ M_2[x := N])$$

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

$$(\lambda y.\ e)[x := N] = \lambda y.\ e[x := N]$$

This will become clear with more examples.

α-reduction

Bound variables are *dummy variables* in the sense that their name is not important. Compare the following functions

$$f(x) = x^2 \qquad\qquad f(t) = t^2$$

The fact that one is written in terms of x, while the other is in terms of t does not change the fact that these functions *do* the same thing: square their input.

In the same way, we are free to rename bound variables in λ-expressions without changing the meaning of the expression. This process is called α-reduction.

Examples

$\lambda x.\ x =_{\alpha} \lambda y.\ y$

$(\lambda x.\ (\lambda y.\ y\ x))\ y =_{\alpha}$

β-reduction

Abstractions ($\lambda x.\ e$) are intended to be interpreted as functions which take in an x and substitute this into free x in the body e of the abstraction.

$$(\lambda x.\ e)\ M =_\beta e[x := M]$$

λ-expressions of the form $(\lambda x.\ e)\ M$ are called β-redex. These are the terms that can be simplified.

Definition: Computation is β-reduction.

Example

Perform β-reduction on the following β-redex

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

Example

Perform β-reduction on the following β-redex

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

Example

Perform β-reduction on the following β-redex

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

Multivariable Functions?

Multivariable functions are abundant in mathematics. Construction rules for λ-expressions only allow for the construction of unary functions. What gives? Partial application, known as Currying, means this actually isn't a problem.

$$f(x, y) = x^2y + y^2x$$

Example

Perform β-reduction on the following β-redex

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

Example

Use α-reduction to relabel this λ-expression so that there are no clashes between bound and free variables.

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

Now reduce the expression using β-reduction.

Reduction Strategies

One λ-expression can consist of multiple β-redex. If this is the case, then different strategies may give different outcomes.

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

If we are to define an automatic model for computation, then we can't have any ambiguity in the process. One needs to decide ahead of time how to deal with these choices.

Two primary strategies are known as call-by-name and call-by-value.

Call-by-name

Once the leftmost β-redex is identified, this strategy calls the leftmost abstraction leaving the input (potentially) unevaluated.

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

Call-by-name is also known as *normal order* or *lazy evaluation*.

Call-by-value

Once the leftmost β-redex is identified, this strategy calls the innermost abstraction.

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

Call-by-value also known as *strict evaluation* or *eager evaluation*.

Normal Form

We say a λ-expression is in normal form if it does not contain any β-redex. If a λ-expression is β-equivalent to a λ-expression in normal form, then it is unique up-to α-reduction.

If a λ-expression has a normal form, then call-by-name evaluation will result in that normal form.

Observation: As a consequence of the above, if you can show that call-by-name evaluation does not terminate, then you may conclude that the λ-expression does not have a normal form.

Example: Call-by-Value

Perform β-reduction on the following β-redex

$$(\lambda y.\ y\ a)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda u.\ u)\ z))$$

Example: Call-by-Name

Perform β-reduction on the following β-redex

$$(\lambda y.\ y\ a)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda u.\ u)\ z))$$

Reduction Graphs

You may come across graphs of λ-expressions for which the different paths through the graph represent different evaluation strategies.

$$(\lambda x.\ 3 \cdot x)\ ((\lambda x.\ x + 1)\ 2)$$

η Abstraction

Notice that the two λ-terms

$$M \qquad\qquad \lambda x.\ M\ x$$

Agree, as functions, on any input. We say λ-terms that agree as functions are "extensionally equivalent". Furthermore, two terms related as above are called eta η equivalent.

$$M =_\eta \lambda x.\ M\ x$$

Syntactic Sugar

Before we start writing real programs in this language we introduce the notation device of "names". These are short hands, so called syntactic sugar, for longer λ-terms. These names help in suppressing details until they are required.

$$I = \lambda x.\ x$$

$$K = \lambda x.\ \lambda y.\ x$$

$$S = \lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z)$$

$$\omega = \lambda x.\ x\ x$$

$$\text{Apply} = \lambda f.\ \lambda a.\ (f\ \ a)$$

We will now talk about how we can encode ideas from logic and arithmetic into the λ-calculus. This requires giving λ-expression encoding for Booleans (TRUE/FALSE), natural numbers, and arithmetic functions and predicates.

Syntactic Sugar

Programs then become strings of these names. These can be desugared, as needed, when computing β-reductions.

$S\ K\ K \equiv_{\mathsf{DS}}$

Conditional Execution

Remember: everything (!) is a λ-expression.

If we want to implement logic in the λ-calculus, then we need λ-expressions that *behave like* TRUE and FALSE, propositional connectives, quantifiers etc.

How do TRUE and FALSE *behave*? One use is conditional branching.

$$\text{COND} :\equiv \lambda c.\ \lambda f.\ \lambda g.\ ((c\ f)\ g)$$

TRUE

COND is an expression with three arguments: c, f, g which should be equivalent to the first argument f if $c =$ TRUE and the second argument g if $c =$ FALSE. It must meet the following specification:

$$\begin{aligned} \text{COND TRUE } f\ g &=_\beta f \\ \text{COND FALSE } f\ g &=_\beta g \end{aligned}$$

Our λ-expression for TRUE needs to ignore the second input g

$$\text{TRUE} = \lambda x.\ \lambda y.\ x$$

TRUE

Perform β-reduction on the following λ-expression until the λ-expression is in normal form.

$$\text{COND TRUE } a\ b = (\lambda c.\ \lambda f.\ \lambda g.\ ((c\ f)\ g))\ \text{TRUE } a\ b$$

FALSE

If the λ-expression for TRUE ignores the second function, then λ-expression for FALSE needs to ignore the first argument.

$$\text{FALSE} =$$

FALSE

Perform β-reduction on the following λ-expression until the λ-expression is in normal form.

$$\text{COND FALSE } a\ b =_\beta \text{FALSE } a\ b$$

Propositional Connectives

TRUE and FALSE should behave appropriately with some implementation in λ-expressions of the propositional connectives: NOT, AND, OR, IMPLIES, NAND, NOR etc.

Since TRUE and FALSE are defined as selectors, the trick is to think about these in terms of selecting the first or second argument.

Example: If the first input to AND is TRUE, then which input should the AND expression return?

NOT

A	NOT A
T	F
F	T

$\text{TRUE} = \lambda x.\ \lambda y.\ x$

$\text{FALSE} = \lambda x.\ \lambda y.\ y$

EXAMPLE

Perform β-reduction on the following λ-expression until the λ-expression is in normal form.

NOT FALSE

Examples

When computing β-reduction on Booleans these redex will arise regularly. So lets simplify them now.

TRUE TRUE

TRUE FALSE

FALSE TRUE

FALSE FALSE

Propositional Connectives

Define the following propositional connectives remembering that TRUE and FALSE are encoded as selectors.

AND

OR

IMPLIES

NAND

NOR

AND

AND is a binary function that returns a Boolean

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

TRUE $= \lambda x.\ \lambda y.\ x$

FALSE $= \lambda x.\ \lambda y.\ y$

AND $= \lambda p.\ \lambda q.$

Example

β-reduce the following λ-expression to normal form.

AND TRUE FALSE

OR

OR is a binary function that returns a Boolean

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

TRUE $= \lambda x.\ \lambda y.\ x$

FALSE $= \lambda x.\ \lambda y.\ y$

OR $= \lambda p.\ \lambda q.$

Example

β-reduce the following λ-expression to normal form.

OR FALSE TRUE

IMPLIES

Implication is a binary function that returns a Boolean

P	Q	$P \to Q$
T	T	T
T	F	F
F	T	T
F	F	T

$\text{TRUE} = \lambda x.\ \lambda y.\ x$

$\text{FALSE} = \lambda x.\ \lambda y.\ y$

$\text{IMPLIES} = \lambda p.\ \lambda q.$

Example

β-reduce the following λ-expression to normal form.

IMPLIES TRUE FALSE

NAND

NAND is a binary function that returns a Boolean

P	Q	NAND P Q
T	T	F
T	F	T
F	T	T
F	F	T

TRUE $= \lambda x.\ \lambda y.\ x$

FALSE $= \lambda x.\ \lambda y.\ y$

NAND $= \lambda p.\ \lambda q.$

NOR

NOR is a binary function that returns a Boolean

P	Q	NOR P Q
T	T	F
T	F	F
F	T	F
F	F	T

TRUE $= \lambda x.\ \lambda y.\ x$

FALSE $= \lambda x.\ \lambda y.\ y$

NOR $= \lambda p.\ \lambda q.$

Propositional Logic Summary

This is a list of encodings for propositional logic into λ-expressions. Note that there are many ways to encode these expressions.

$$\text{COND} :\equiv \lambda c.\ \lambda f.\ \lambda g.\ c\ f\ g$$

$$\text{TRUE} :\equiv \lambda x.\ \lambda y.\ x$$

$$\text{FALSE} :\equiv \lambda x.\ \lambda y.\ y$$

$$\text{NOT} :\equiv \lambda p.\ \text{FALSE TRUE}$$

$$\text{AND} :\equiv \lambda p.\ \lambda q.\ p\ q\ p$$

$$\text{OR} :\equiv \lambda p.\ \lambda q.\ p\ p\ q$$

$$\text{IMPLIES} :\equiv \lambda p.\ \lambda q.\ \text{OR}\ (\text{NOT}\ p)\ q$$

$$\text{NAND} :\equiv$$

$$\text{NOR} :\equiv$$

Church Numerals

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

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

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 the λ-term:

$$n\ f\ a$$

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

Programming in λ-Calculus

We will now write λ-expressions (programs!) to do arithmetic on Church encoded natural numbers. It is important to remember that 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 write λ-expressions for fundamental arithmetic functions and predicates on Church numerals.

Arithmetic Functions: SUCC, SUM, MULT, EXP, PRED, SUB.

Arithmetic Predicates: ZERO?, GREATER?, EQUAL? etc.

We will adopt the convention of writing names of predicates with a "?" at the end. This helps readability of programs.

Encoding Arithmetic Functions

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

Unary functions on Church numerals will always start

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

Binary functions on Church numerals will always start

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

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.

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

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.

MULT $= \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.$

Example

MULT TWO TWO

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 of Church encoded numerals.

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

Example

β-reduce the following expression to normal form

EXP THREE TWO

ZERO?

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

This predicate should satisfy the following specification:

$$\begin{aligned} \text{ZERO? ZERO} &=_\beta \text{TRUE} \\ \text{ZERO? ONE} &=_\beta \text{FALSE} \\ &\vdots \end{aligned}$$

ZERO? $= \lambda m.$

Parity

Write a λ-term that reduces to TRUE if the input is even, and FALSE if the input is odd.

EVEN? $:\equiv\ \lambda n.$

Parity

Write a λ-term that reduces to TRUE if the input is odd, and FALSE if the input is even.

ODD? $:\equiv \ \lambda n.$

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.

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

Example

PAIR ONE TWO FST $=_\beta$ ONE

PAIR ONE TWO SND $=_\beta$ TWO

PAIR ONE (PAIR TWO THREE) SND $=_\beta$ 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.

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

Example

PRED ONE

SUB

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

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

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?

$\text{EQUAL?} = \lambda m.\ \lambda n.$

Example

EQUAL? ONE ZERO

Summary

Arithmetic Functions

$$\begin{aligned}
\text{SUCC} &:\equiv \lambda n.\ \lambda u.\ \lambda v.\ u(n\ u\ v) \\
\text{SUM} &:\equiv \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.\ m\ u\ (n\ u\ v) \\
\text{MULT} &:\equiv \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.\ m\ (n\ u)\ v \\
\text{EXP} &:\equiv \lambda e.\ \lambda b. eb \\
\text{PRED} &:\equiv \lambda n.\ (n\ \Psi\ (\text{PAIR ZERO ZERO}))\ \text{SND} \\
\text{SUB} &:\equiv \lambda m.\ \lambda n.\ \lambda u.\ \lambda v.\ (n\ \text{PRED}\ m)\ u\ v
\end{aligned}$$

Arithmetic Predicates

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

Recursion?

We have implemented programs in the lambda calculus to carry out arithmetic like SUM, MULT, and PRED. These make use of concatenating strings of "successor" function applications.

However we know from Peano Arithmetic that these functions all have *recursive* definitions. Can we implement these arithmetic functions recursively in the lambda calculus?

Recursion will allows us to write programs that search for solutions to certain predicates. As well as define bounded quantification predicates: for all x less than n, and there exists an x less than n. Programs that we might be more used to writing with while-loops in modern, imperative, programming languages like C and Python.

Recursive SUM

Recall that axioms PA3 and PA4 define $(+)$ SUM as:

$$\text{SUM } a\ b = \begin{cases} a & b = 0 \\ \text{SUCC (SUM } a \text{ (PRED } B)) & \text{Otherwise} \end{cases}$$

This requires conditional branching, checking whether a number is zero, calculating the successor and the predecessor. We have seen the lambda encodings of these processes.

However, the lambda abstraction syntax does not allow for a function to refer to itself by name. This is because functions do not have names in this syntax.

We give λ-terms names, but only once they're defined.

Towards Recursive SUM

Self-reference is not part of the λ-calculus syntax. However, abstracting over SUM and the numeral inputs gives us the following valid λ expression:

$$\text{GO} := \lambda s.\ \lambda a.\ \lambda b.\ \text{COND}\ (\text{ZERO?}\ b)\ a\ (\text{SUCC}\ (s\ a\ (\text{PRED}\ b)))$$

We can manually pass this function to itself to mimic recursion.

GO GO

$$=_\beta \lambda a.\ \lambda b.\ \text{COND}\ (\text{ZERO?}\ b)\ a\ (\text{SUCC}\ (\text{GO}\ a\ (\text{PRED}\ b)))$$

This gives us a function to which we can pass numerals.

Does it return the sum?

Towards Recursive SUM

GO GO

$$=_\beta \lambda a.\ \lambda b.\ \text{COND (ZERO? } b)\ a\ (\text{SUCC (GO } a \text{ (PRED } b)))$$

If $b =$ ZERO, then this β-reduces to a. In this case GO GO returns the correct sum.

However, if $b \neq$ ZERO, then the second condition gets executed.

$$\text{GO GO ONE ONE} =_\beta \text{SUCC (GO ONE (PRED ONE))}$$

At some point ONE is passed to the $\lambda s.$ abstraction in GO. This will lead to nonsense. In this case we see that GO GO will not return the correct sum of TWO.

Our recursion is not deep enough.

Towards Recursive SUM

Pass the previous iteration back through GO.

GO (GO GO)

$=_\beta \lambda a.\ \lambda b.\ \mathrm{COND}(\mathrm{ZERO?}\ b)\ a\ (\mathrm{SUCC}\ ((\mathrm{GO}\ \mathrm{GO})\ a\ (\mathrm{PRED}\ b)))$

As before this is correct for $b = \mathrm{ZERO}$.

What happens now for $b = \mathrm{ONE}$?

Towards Recursive SUM

If the second input is ONE we can do the following reduction:

$$
\begin{aligned}
&\text{GO (GO GO) ONE ONE} \\
&=_\beta \text{SUCC(GO GO ONE (PRED ONE))} \\
&=_\beta \text{SUCC(GO GO ONE ZERO)} \\
&=_\beta \text{SUCC (ONE)} \\
&=_\beta \text{TWO}
\end{aligned}
$$

However, if the second input were greater than ONE, then the reduction would reduce to nonsense for the same reason as before.

Towards Recursive SUM

This suggests that if we want to add numbers using recursion, then we look at the second input and choose how many times we pass GO to itself before passing the numeral inputs.

> GO GO
>
> GO (GO GO)
>
> GO (GO (GO GO))
>
> GO (GO (GO (GO GO)))
>
> ⋮

REALLY!?

Is there not a way to do this automatically?

Y-Combinator

Consider this magic passed down from Haskell B. Curry

$$\mathsf{Y} := \ \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Applying this to a function gives exactly what we need:

Y-Combinator

Consider this magic passed down from Haskell B. Curry

$$\mathsf{Y} := \ \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Applying this to a function gives exactly what we need:

$$\begin{aligned}
\mathsf{Y}\ g &= (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g \\
&=_\beta (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)) \\
&=_\beta g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))) \\
&=_\beta g\ (\mathsf{Y}\ g) \\
&=_\beta g\ (g\ (\mathsf{Y}\ g)) \\
&=_\beta g\ (g\ (g\ (\mathsf{Y}\ g))) \\
&\ \vdots
\end{aligned}$$

Recursive SUM

This suggests the following definition:

$$\text{SUM} :\equiv \text{Y GO}$$

Following a few steps of β reduction yields:

$$\begin{aligned}\text{SUM} :\equiv{}& \text{Y GO}\\ &=_\beta \text{GO (Y GO)}\\ &=_\beta \lambda a.\ \lambda b.\ \text{COND (ZERO? } b)\ a\ (\text{SUCC (Y GO } a\ (\text{PRED } b)))\\ &\equiv \lambda a.\ \lambda b.\ \text{COND (ZERO? } b)\ a\ (\text{SUCC (SUM } a\ (\text{PRED } b)))\end{aligned}$$

This is a function to which one can pass Church numerals and it will compute the sum of the two inputs by recursively calling itself!

Example

SUM ONE TWO

Y Combinator Recursion

This approach will yield recursive implementations of other functions we already know to have recursive definitions [**pierce**].

Given a function, FUNK, known to be recursive follow the steps above to determine an implementation of FUNK in the λ-calculus:

- Give the "named" recursive definition
- Write GO by abstracting over the name and inputs
- Define FUNK = Y GO

GO get the FUNK.

Higher Order Procedures

We have seen that there is very little distinction between data and procedure. In the recursive definition of SUM we passed the "data" of GO to the procedure Y. We passed procedures, to other procedures to generate yet more procedures. We call these procedures that take other procedures as input "higher order procedures".

To further illustrate the power of the λ-calculus - to give further evidence to Church's Thesis - we will no spend some time developing more higher order procedures in the untyped lambda calculus:

Search μ and bounded search $\overline{\mu}$

DIVIDES? COMPOSITE? PRIME?

ACCumulate and FILteredACCumulate.

Unbounded Search

Suppose $P?(x)$ is a unary predicate that can be implemented with a λ-expression. We can define a recursive algorithm to search for the smallest natural number that satisfies $P?(x)$.

$\mu = \lambda$

Suppose $P?(x)$ is a unary predicate that can be implemented with a λ-expression. We can define a recursive algorithm to search for the smallest natural number that satisfies $P?(x)$.

First define the following helper-function:

$$\mathsf{GO} :\equiv \lambda s.\ \lambda x.\ \mathsf{COND}\ P?(x)\ x\ (s\ (\mathsf{SUCC}\ x))$$

We may define the recursive search by:

$$\mu :\equiv\ \mathsf{Y}\ \mathsf{GO}$$

To compute the smallest natural number that satisfies $P?(x)$ we run the program:

$$\mu\ \mathsf{ZERO}$$

Bounded Search $\overline{\mu}$

If $P?(x)$ is a unary predicate, then we can search for a solution in a bounded interval using the following procedure:

Arithmetic Predicates

DIVIDES? $= \lambda n.\ \lambda d.\ \overline{\mu}\ (\lambda k.\ =?\ n$ (MUL $k\ d$)) 1 n

COMPOSITE? $= \lambda x.\ \overline{\mu}$ (DIVIDES? x) 2 (PRED x)

PRIME? $= \lambda x.$ NOT(COMPOSITE? x)

Same, But Different

No assignment with mutable state. No specific syntax for writing for- or while-loops. “Just” recursion. How do we do all the things we’re used to when writing programs in languages like C or Python?

Loops with Recursion

Suppose we want to sum up the integers in the list

$$1, 2, 3, 4, ..., 10$$

We follow the named-description on the previous slide and abstract over the inputs and function name to get the following helper:

$$\begin{aligned} \text{GO} :\equiv \lambda s.\ \lambda a.\ \lambda l.\ \lambda u.\ \text{COND}\ & (>?\ l\ u) \\ & a \\ & (s\ (\text{SUM}\ a\ l)\ (\text{SUCC}\ l)\ u) \end{aligned}$$

Combining this with the Y combinator gives a procedure to ACCumulate the integers from l up to u:

$$\text{ACC} :\equiv \text{Y GO}$$

In order to compute the sum you pass the starting value of the sum (ZERO) and the lower and upper bound of the interval.

Example

Suppose we want to calculate the sum of the integers $[1, 10]$.

We write the following procedure with ACCumulate:

$$
\begin{aligned}
&\text{ACC ZERO ONE TEN} \\
&=_\beta \text{ COND } (>\text{? ONE TEN}) \text{ ZERO} \\
&\qquad \text{(ACC (SUM ZERO ONE) (SUCC ONE) TEN)} \\
&=_\beta \text{ ACC (SUM ZERO ONE) (SUCC ONE) TEN} \\
&=_\beta \text{ ACC ONE TWO TEN} \qquad \vdots
\end{aligned}
$$

Filtered Accumulate

Question: How would you change the ACC program to only sum those elements which satisfy $P?(x)$ a unary predicate?

Combinator Calculus

As often happens two people independently formulated the Combinator Calculus (CL): Moses Shönfinkel and Haskell Curry. Shönfinkel was motivated to remove quantifiers and bound variables from predicate logic - as we have seen substitution and the issue of free/bound variables is subtle. We also see a similar problem in the λ-calculus with variables bound by abstractions.

If this could be avoided, then reduction would be made simpler!

Grammar of Combinator Calculus

SKI combinator calculus is made up of the constants S, K, I and the single application constructor.

Combinators are subject to the following reduction rules:

$$\begin{aligned} S\ t\ u\ v\ &=_\beta t\ v\ (u\ v) \\ K\ t\ u\ &=_\beta t \\ I\ x &=_\beta x \end{aligned}$$

Example

Show $SKK = I$ extensionally.

Embedding CL into λ

We can embed CL into λ-calculus as follows:

$$\begin{aligned} I &= \lambda x.\ x \\ K &= \lambda x.\ \lambda y.\ x \\ S &= \lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z) \end{aligned}$$

Note that these definitions are extensional equivalences. Each CL term and the corresponding λ-term are equal as functions; that is, they compute the same output for the same input.

Emedding λ into CL

In fact one can embed λ-calculus into CL. In this case the embedding is an extensional equivalence. We define the translation procedure T from λ to CL recursively as follows:

This translation preserves extensional equality.

Complete Bases

We say $\{S, K, I\}$ is a complete base because these combinators generate all λ-terms extensionally.

S K I

S K

B C K W

I J K

X

Descendants of λ-Calculi

If you enjoy programming in this fashion, then you might enjoy working in one of the modern descendants of this approach to computing. Some examples are:

- Racket (LISP dialect)
- Haskell
- O'Caml

Racket is easy to get started with. Download the IDE Dr Racket. There are a number of good places to learn LISP languages. My favourite is the textbook *Structure and Interpretation of Computer Programs* and the associated lectures available on YouTube.

Haskell uses a typed version of the λ-calculus. Adding types to the lambda calculus is the next topic of the course.

Further Reading

These lecture slides were prepared with the aid of the following references. They can be consulted for further detail on the topics.

- Stanford Encyclopedia of Philosophy.
- Type Theory and Functional Programming, *Simon Thompson*.
- Types and Programming Languages, *Benjamin Pierce.*