CMSC 330: Organization of Programming Languages

Lambda Calculus
Turing Machine

Infinite Tape

1 0 0 0 1 1 1 1 0

Read / Write Head

Control Unit
State: Y

START

HALT

2

3

4

a; a, R
b; b, R

e; e, R
b; b, R

a; a, R
a; a, R

a; a, R
a; a, R

b; b, R
b; b, R
Turing Completeness

- Turing machines are the most powerful description of computation possible
  - They define the Turing-computable functions
- A programming language is Turing complete if
  - It can map every Turing machine to a program
  - A program can be written to emulate a Turing machine
  - It is a superset of a known Turing-complete language
- Most powerful programming language possible
  - Since Turing machine is most powerful automaton
Programming Language Expressiveness

- So what language features are needed to express all computable functions?
  - What’s a minimal language that is Turing Complete?
- Observe: some features exist just for convenience
  - Multi-argument functions
    - foo (a, b, c)
      - Use currying or tuples
  - Loops
    - while (a < b) …
      - Use recursion
  - Side effects
    - a := 1
      - Use functional programming pass “heap” as an argument to each function, return it when with function’s result:
        - effectful : ‘a → ‘s → (‘s * ‘a)
Programming Language Expressiveness

- It is not difficult to achieve Turing Completeness
  - Lots of things are ‘accidentally’ TC
- Some fun examples:
  - x86_64 `mov` instruction
  - Minecraft
  - Magic: The Gathering
  - Java Generics
- There’s a whole cottage industry of proving things to be TC
- But: What is a “core” language that is TC?
Lambda Calculus ($\lambda$-calculus)

- Proposed in 1930s by
  - Alonzo Church  
    (born in Washington DC!)

- Formal system
  - Designed to investigate functions & recursion
  - For exploration of foundations of mathematics

- Now used as
  - Tool for investigating computability
  - Basis of functional programming languages
    - Lisp, Scheme, ML, OCaml, Haskell…
Why Study Lambda Calculus?

- It is a “core” language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, …
- Plus, higher-order, anonymous functions (aka _lambdas_) are now very popular!
  - C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), … (and functional languages like OCaml, Haskell, F#, …)
  - Excel, as of 2021!
Lambda Calculus Syntax

- A lambda calculus **expression** is defined as

  \[ e ::= x \quad \text{variable} \]

  \[
  | \quad \lambda x. e \quad \text{abstraction (fun def)} \\
  | \quad e \ e \quad \text{application (fun call)}
  \]

- This grammar describes ASTs; not for parsing - ambiguous!
- Lambda expressions also known as lambda **terms**

- \( \lambda x. e \) is like \((\text{fun } x \rightarrow e)\) in OCaml

That’s it! Nothing but higher-order functions
Three Conventions

- Scope of \( \lambda \) extends as far right as possible
  - Subject to scope delimited by parentheses
  - \( \lambda x. \lambda y. x \, y \) is same as \( \lambda x.(\lambda y.(x \, y)) \)

- Function application is left-associative
  - \( x \, y \, z \) is \( (x \, y) \, z \)
  - Same rule as OCaml

- As a convenience, we use the following “syntactic sugar” for local declarations
  - \( \text{let } x = e1 \text{ in } e2 \) is short for \( (\lambda x.e2) \, e1 \)
Quiz #1

\( \lambda x. (y \ z) \) and \( \lambda x. y \ z \) are equivalent

A. True
B. False
Quiz #1

\( \lambda x. (y z) \) and \( \lambda x. y z \) are equivalent

A. True
B. False
Quiz #2

This term is equivalent to which of the following?

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

A. \((\lambda x. x) \ (a \ b)\)
B. \(((\lambda x. x) \ a) \ b)\)
C. \(\lambda x. (x \ (a \ b))\)
D. \((\lambda x. ((x \ a) \ b))\)
Quiz #2

This term is equivalent to which of the following?

\[ \lambda x. x \ a \ b \]

A. \((\lambda x. x) \ (a \ b)\)
B. \(((\lambda x. x) \ a) \ b)\)
C. \(\lambda x. \ (x \ (a \ b))\)
D. \((\lambda x. ((x \ a) \ b))\)
Lambda Calculus Semantics

- Evaluation: All that’s involved are function calls
  \((\lambda x. e_1) \ e_2\)
  - Evaluate \(e_1\) with \(x\) replaced by \(e_2\)

- This application is called beta-reduction
  - \((\lambda x. e_1) \ e_2 \rightarrow e_1[x:=e_2]\)
    - \(e_1[x:=e_2]\) is \(e_1\) with occurrences of \(x\) replaced by \(e_2\)
    - This operation is called substitution
      - Replace formals with actuals
      - Instead of using environment to map formals to actuals
  - We allow reductions to occur anywhere in a term
    - Order reductions are applied does not affect final value!

- When a term cannot be reduced further it is in beta normal form
Beta Reduction Example

\[ (\lambda x.\lambda z.x\ z)\ y \]
\[ \rightarrow (\lambda x.(\lambda z.(x\ z)))\ y \] \hspace{1cm} // since \( \lambda \) extends to right
\[ \rightarrow (\lambda x.(\lambda z.(x\ z)))\ y \] \hspace{1cm} // apply \((\lambda x.e1)\ e2 \rightarrow e1[x:=e2]\)
\[ \hspace{1cm} // \text{where } e1 = \lambda z.(x\ z), \ e2 = y \]
\[ \rightarrow \lambda z.(y\ z) \] \hspace{1cm} // final result

- Equivalent OCaml code
  - \((\text{fun } x -> (\text{fun } z -> (x\ z)))\ y \rightarrow \text{fun } z -> (y\ z)\)
Two Varieties

- There are two common variants of big-step semantics
  - *Eager* evaluation (aka *strict*, or *call by value*)
  - *Lazy* evaluation (aka *call by name*)
Call by Value

- Before doing a beta reduction, we make sure the argument cannot, itself, be further evaluated
- This is known as call-by-value (CBV)
  - This is the Eager big step approach

\[
\begin{align*}
\text{e}_1 \rightarrow e_3 \\
\text{e}_1 \text{e}_2 \rightarrow e_3 \text{ e}_2 \\
\text{e}_2 \rightarrow e_3 \\
\text{e}_1 \text{e}_2 \rightarrow e_1 \text{e}_3
\end{align*}
\]

\[
\begin{align*}
e &= (\lambda x. e \text{e}_2) \text{ or } e = y \\
(\lambda x. e_1) e \rightarrow e_1[x:=e]
\end{align*}
\]
Beta Reductions (CBV)

- $(\lambda x. x) \, z \rightarrow z$
- $(\lambda x. y) \, z \rightarrow y$
- $(\lambda x. x \, y) \, z \rightarrow z \, y$
  - A function that applies its argument to $y$
Beta Reductions (CBV)

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

- $(\lambda x. \lambda y. x \ y) \ z \rightarrow \lambda y. z \ y$
  - A curried function of two arguments
  - Applies its first argument to its second

- $(\lambda x. \lambda y. x \ y) \ (\lambda z. zz) \ x \rightarrow (\lambda y. (\lambda z. zz)y) x \rightarrow (\lambda z. zz) x \rightarrow x \ x$
Quiz #3

(\lambda x. y) z can be beta-reduced to

A. y
B. y z
C. z
D. cannot be reduced
Quiz #3

\((\lambda x. y) \ z\) can be beta-reduced to

A. \(y\)
B. \(y \ z\)
C. \(z\)
D. cannot be reduced
Quiz #4

Which of the following reduces to $\lambda z. z$?

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

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

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

d) $(\lambda y. \lambda x. z) z (\lambda z. z)$
Quiz #4

Which of the following reduces to $\lambda z. \ z$?

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

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

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

d) $(\lambda y. \lambda x. \ z) \ z \ (\lambda z. \ z)$
Call by Name

- Instead of the CBV strategy, we can specifically choose to perform beta-reduction *before* we evaluate the argument.
- This is known as **call-by-name (CBN)**.
  - This is the Lazy small-step approach.

\[
\begin{align*}
e_1 & \rightarrow e_3 \\
e_1 \ e_2 & \rightarrow e_3 \ e_2 \\
(\lambda x. e_1) \ e_2 & \rightarrow e_1[x:=e_2]
\end{align*}
\]
CBN Reduction

- **CBV**
  - \((\lambda z.z)((\lambda y.y) \ x) \rightarrow (\lambda z.z) \ x \rightarrow x\)

- **CBN**
  - \((\lambda z.z)((\lambda y.y) \ x) \rightarrow (\lambda y.y) \ x \rightarrow x\)
Beta Reductions (CBN)

\[(\lambda x. x (\lambda y. y)) (u \ r) \rightarrow \]

\[(\lambda x. (\lambda w. x w)) (y \ z) \rightarrow \]
Beta Reductions (CBN)

\[(\lambda x. x (\lambda y. y)) \ (u \ r) \to (u \ r) \ (\lambda y. y)\]

\[(\lambda x. (\lambda w. x \ w)) \ (y \ z) \to (\lambda w. (y \ z) \ w)\]
 Lambda calculus uses **static scoping**

Consider the following

- \((\lambda x.x \ (\lambda x.x)) \ z \to \ ?\)
  - The rightmost “x” refers to the second binding
- This is a function that
  - Takes its argument and applies it to the identity function

This function is “the same” as \((\lambda x.x \ (\lambda y.y))\)

- Renaming bound variables consistently preserves meaning
  - This is called **alpha-renaming** or **alpha conversion**
- Ex. \(\lambda x.x = \lambda y.y = \lambda z.z\)  \(\lambda y.\lambda x.y = \lambda z.\lambda x.z\)
Quiz #5

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) $${\lambda} y. \ y \ y$$
b) $${\lambda} z. \ y \ z$$
c) $$(\lambda x. \lambda z. x \ z) \ y$$
d) $$(\lambda x. \lambda y. x \ y) \ z$$
Quiz #5

Which of the following expressions is alpha equivalent to (alpha-converts from)

\[(\lambda x. \lambda y. x y) \, y\]

a) \(\lambda y. y \, y\)

b) \(\lambda z. y \, z\)

c) \((\lambda x. \lambda z. x \, z) \, y\)

d) \((\lambda x. \lambda y. x \, y) \, z\)
Defining Substitution

- Use recursion on structure of terms
  - $x[x:=e] = e$  // Replace $x$ by $e$
  - $y[x:=e] = y$  // $y$ is different than $x$, so no effect
  - $(e_1 e_2)[x:=e] = (e_1[x:=e]) (e_2[x:=e])$  // Substitute both parts of application
  - $(\lambda x.e'[x:=e]) = \lambda x.e'$
  
  - $(\lambda y.e'[x:=e]) = ?$
    - $(\lambda y.(e'[x:=e]))$  If $x \not\in (\text{fvs } e)$
      - $(\lambda y. x y) z = (\lambda y. z y)$
    - $(\lambda y.(e'[x:=e]))$  alpha-convert $e'$ if $x \in (\text{fvs } e)$
      - $(\lambda y. x y) y = (\lambda z. x z) y = \lambda z. y z$
Variable Capture

How about the following?

- \((\lambda x.\lambda y.x \ y) \ y \rightarrow ?\)
- When we replace \(y\) inside, we don’t want it to be captured by the inner binding of \(y\), as this violates static scoping
- I.e., \((\lambda x.\lambda y.x \ y) \ y \neq \lambda y.y \ y\)

Solution

- \((\lambda x.\lambda y.x \ y)\) is “the same” as \((\lambda x.\lambda z.x \ z)\)
  - Due to alpha conversion
- So alpha-convert \((\lambda x.\lambda y.x \ y) \ y\) to \((\lambda x.\lambda z.x \ z) \ y\) first
  - Now \((\lambda x.\lambda z.x \ z) \ y \rightarrow \lambda z.y \ z\)
Completing the Definition of Substitution

- Recall: we need to define $(\lambda y. e')[x:=e]$
  - We want to avoid capturing (free) occurrences of $y$ in $e$
  - Solution: alpha-conversion!
    - Change $y$ to a variable $w$ that does not appear in $e'$ or $e$
      (Such a $w$ is called fresh)
    - Replace all occurrences of $y$ in $e'$ by $w$.
    - Then replace all occurrences of $x$ in $e'$ by $e$!

- Formally:
  $$(\lambda y. e')[x:=e] = \lambda w.((e' [y:=w]) [x:=e]) \text{ (}w\text{ is fresh}))$$
Whenever we do a step of beta reduction

- \((\lambda x. e_1) \ e_2 \rightarrow e_1[x:=e_2]\)
- We must alpha-convert variables as necessary
- Sometimes performed implicitly (w/o showing conversion)

Examples

- \((\lambda x. \lambda y. x \ y) \ y = (\lambda x. \lambda z. x \ z) \ y \rightarrow \lambda z. y \ z \quad // \ y \rightarrow z\)
- \((\lambda x. x \ (\lambda x.x)) \ z = (\lambda y. y \ (\lambda x.x)) \ z \rightarrow z \ (\lambda x.x) \quad // \ x \rightarrow y\)
Quiz #6

Beta-reducing the following term produces what result?

\[(\lambda x. x \, \lambda y. y \, x) \, y\]

A. \( y \, (\lambda z. z \, y) \)
B. \( z \, (\lambda y. y \, z) \)
C. \( y \, (\lambda y. y \, y) \)
D. \( y \, y \)
Quiz #6

Beta-reducing the following term produces what result?

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

A. \(y (\lambda z. z y)\)
B. \(z (\lambda y. y z)\)
C. \(y (\lambda y. y y)\)
D. \(y y\)
Quiz #7

Beta reducing the following term produces what result?

\[ \lambda x. (\lambda y. y y) w z \]

a) \[ \lambda x. w w z \]

b) \[ \lambda x. w z \]

c) \[ w z \]

d) Does not reduce
Quiz #7

Beta reducing the following term produces what result?

\[ \lambda x. (\lambda y. y y) \; w \; z \]

a) \( \lambda x. w \; w \; z \)

b) \( \lambda x. w \; z \)

c) \( w \; z \)

d) Does not reduce
Lambda Calc, Impl in OCaml

\[ e ::= x \mid \lambda x.e \mid e e \]

\[
\begin{align*}
y & \quad \text{Var "y"} \\
\lambda x.x & \quad \text{Lam ("x", Var "x")} \\
\lambda x.\lambda y.x y & \quad \text{Lam ("x", (Lam("y", App (Var "x", Var "y")))))} \\
(\lambda x.\lambda y.x y) \lambda x.x x & \quad (\text{Lam ("x", Lam ("y", App (Var "x", Var "y"))}, \\
& \quad \text{Lam ("x", App (Var "x", Var "x")))})
\end{align*}
\]

\[
\begin{align*}
type \ id & = \text{string} \\
type \ exp & = \text{Var of id} \\
& \mid \text{Lam of id * exp} \\
& \mid \text{App of exp * exp}
\end{align*}
\]
Quiz #8

What is this term’s AST?

\( \lambda x.x x x \)

A. App \( (\text{Lam ("x", Var "x"), Var "x"}) \)
B. Lam \( (\text{Var "x", Var "x", Var "x"}) \)
C. Lam \( ("x", \text{App (Var "x",Var "x"))} \)
D. App \( (\text{Lam ("x", App ("x", "x")))} \)

type id = string
type exp =
| Var of id
| Lam of id * exp
| App of exp * exp
Quiz #8

What is this term’s AST?

\[ \lambda x.x \ x \ x \]

A. App (Lam ("x", Var "x"), Var "x")
B. Lam (Var "x", Var "x", Var "x")
C. Lam ("x", App (Var "x", Var "x"))
D. App (Lam ("x", App ("x", "x")))
The Power of Lambdas

- To give a sense of how one can encode various constructs into LC we’ll be looking at some concrete examples:
  - Let bindings
  - Booleans
  - Pairs
  - Natural numbers & arithmetic
  - Looping
Let bindings

- Local variable declarations are like defining a function and applying it immediately (once):
  - \( \text{let } x = e_1 \text{ in } e_2 = (\lambda x. e_2) \ e_1 \)

- Example
  - \( \text{let } x = (\lambda y. y) \text{ in } x \ x = (\lambda x. x \ x) (\lambda y. y) \)

where

\( (\lambda x. x \ x) (\lambda y. y) \rightarrow (\lambda x. x \ x) (\lambda y. y) \rightarrow (\lambda y. y) (\lambda y. y) \rightarrow (\lambda y. y) \)
Booleans

- Church’s encoding of mathematical logic
  - true = λx.λy.x
  - false = λx.λy.y
  - if a then b else c
    - Defined to be the expression: a b c

- Examples
  - if true then b else c = (λx.λy.x) b c → (λy.b) c → b
  - if false then b else c = (λx.λy.y) b c → (λy.y) c → c
Other Boolean operations

- **not** = \( \lambda x. x \) false true
  - not \( x \) = \( x \) false true = if \( x \) then false else true
  - not true \( \rightarrow (\lambda x. x \) false true\) true \( \rightarrow (true \) false true\) \( \rightarrow \) false

- **and** = \( \lambda x. \lambda y. x \) y false
  - and \( x \) y = if \( x \) then \( y \) else false

- **or** = \( \lambda x. \lambda y. x \) true \( y \)
  - or \( x \) y = if \( x \) then true else \( y \)

Given these operations

- Can build up a logical inference system
Quiz #9

What is the lambda calculus encoding of \( \text{xor} \ x \ y \)?

- \( \text{xor true true} = \text{xor false false} = \text{false} \)
- \( \text{xor true false} = \text{xor false true} = \text{true} \)

\[
\begin{align*}
\text{true} &= \lambda x.\lambda y.x \\
\text{false} &= \lambda x.\lambda y.y \\
\text{if } a \text{ then } b \text{ else } c &= a \ b \ c \\
\text{not} &= \lambda x.\text{x} \text{ false} \text{ true}
\end{align*}
\]
Quiz #9

What is the lambda calculus encoding of $\text{xor } x \ y$?

- $\text{xor true true} = \text{false}$
- $\text{xor false false} = \text{false}$
- $\text{xor true false} = \text{true}$
- $\text{xor false true} = \text{true}$

- $\text{x x y}$
- $\text{x (y true false) y}$
- $\text{x (y false true) y}$
- $\text{y x y}$

true = $\lambda x.\lambda y. x$
false = $\lambda x.\lambda y.y$
if $a$ then $b$ else $c = a \ b \ c$
not = $\lambda x.x$ false true
Pairs

- Encoding of a pair \( a, b \)
  - \( (a,b) = \lambda x.\text{if } x \text{ then } a \text{ else } b \)
  - \( \text{fst} = \lambda f. f \text{ true} \)
  - \( \text{snd} = \lambda f. f \text{ false} \)

- Examples
  - \( \text{fst} (a,b) = (\lambda f. f \text{ true}) \ (\lambda x.\text{if } x \text{ then } a \text{ else } b) \rightarrow \) 
    \( (\lambda x.\text{if } x \text{ then } a \text{ else } b) \) true \rightarrow 
    if true then a else b \rightarrow a
  - \( \text{snd} (a,b) = (\lambda f. f \text{ false}) \ (\lambda x.\text{if } x \text{ then } a \text{ else } b) \rightarrow \) 
    \( (\lambda x.\text{if } x \text{ then } a \text{ else } b) \) false \rightarrow 
    if false then a else b \rightarrow b
Natural Numbers (Church* Numerals)

- Encoding of non-negative integers
  - 
  
  - \( 0 = \lambda f.\lambda y. y \)
  - \( 1 = \lambda f.\lambda y. f\ y \)
  - \( 2 = \lambda f.\lambda y. f\ (f\ y) \)
  - \( 3 = \lambda f.\lambda y. f\ (f\ (f\ y)) \)

  i.e., \( n = \lambda f.\lambda y. \text{<apply}\ f\ \text{n times to y>} \)

  - Formally: \( n+1 = \lambda f.\lambda y. f\ (n\ f\ y) \)

*(Alonzo Church, of course)*
Quiz #10

What OCaml type could you give to a Church-encoded numeral?

- (‘a -> ‘b) -> ‘a -> ‘b
- (‘a -> ‘a) -> ‘a -> ‘a
- (‘a -> ‘a) -> ‘b -> int
- (int -> int) -> int -> int

\[ n = \lambda f.\lambda y.\langle \text{apply } f \ n \ \text{times to } y \rangle \]
Quiz #10

\[ n = \lambda f.\lambda y.\langle \text{apply } f \ n \ \text{times to } y \rangle \]

What OCaml type could you give to a Church-encoded numeral?

- \((\text{'}a -\rightarrow \text{'}b) -\rightarrow \text{'}a -\rightarrow \text{'}b\)
- \((\text{'}a -\rightarrow \text{'}a) -\rightarrow \text{'}a -\rightarrow \text{'}a\)
- \((\text{'}a -\rightarrow \text{'}a) -\rightarrow \text{'}b -\rightarrow \text{int}\)
- \((\text{\text{int} -\rightarrow \text{int}}) -\rightarrow \text{\text{int} -\rightarrow \text{int}}\)
Operations On Church Numerals

- **Successor**
  - \( \text{succ} = \lambda z. \lambda f. \lambda y. f \ (z \ f \ y) \)
  - \( 0 = \lambda f. \lambda y. y \)
  - \( 1 = \lambda f. \lambda y. f \ y \)

- **Example**
  - \( \text{succ} \ 0 = \)
    - \( (\lambda z. \lambda f. \lambda y. f \ (z \ f \ y)) \ (\lambda f. \lambda y. y) \rightarrow \)
    - \( \lambda f. \lambda y. f \ ((\lambda f. \lambda y. y) \ f \ y) \rightarrow \)
    - \( \lambda f. \lambda y. f \ ((\lambda y. y) \ y) \rightarrow \)
    - \( \lambda f. \lambda y. f \ y \rightarrow \) \( \lambda f. \lambda y. f \ y \)
  - \( = 1 \)
  - Since \( (\lambda x. y) \ z \rightarrow y \)
Operations On Church Numerals (cont.)

- **IsZero?**
  - iszero = λz.z (λy.false) true
    
    This is equivalent to λz.((z (λy.false)) true)

- **Example**
  - 0 = λf.λy.y
    - iszero 0 = 
      
      (λz.z (λy.false) true) (λf.λy.y) →
      
      (λf.λy.y) (λy.false) true →
      
      (λy.y) true → Since (λx.y) z → y
      
      true
Arithmetic Using Church Numerals

- If M and N are numbers (as λ expressions)
  - Can also encode various arithmetic operations

- Addition
  - \( M + N = \lambda f.\lambda y.M \ f \ (N \ f \ y) \)
  - Equivalently: \( + = \lambda M.\lambda N.\lambda f.\lambda y.M \ f \ (N \ f \ y) \)
  - In prefix notation (+ M N)

- Multiplication
  - \( M * N = \lambda f.M \ (N \ f) \)
  - Equivalently: \( * = \lambda M.\lambda N.\lambda f.\lambda y.M \ (N \ f) \ y \)
  - In prefix notation (* M N)
Arithmetic (cont.)

- Prove $1+1 = 2$
  - $1+1 = \lambda x.\lambda y. (1 \ x) \ (1 \ x \ y) =$
  - $\lambda x.\lambda y. ((\lambda f.\lambda y. f \ y) \ x) \ (1 \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. (\lambda y. x \ y) \ (1 \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ (1 \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ ((\lambda f.\lambda y. f \ y) \ x \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ ((\lambda y. x \ y) \ y) \rightarrow$
  - $\lambda x.\lambda y. x \ (x \ y) = 2$

- With these definitions
  - Can build a theory of arithmetic

- $1 = \lambda f.\lambda y. f \ y$
- $2 = \lambda f.\lambda y. f \ (f \ y)$
Arithmetic Using Church Numerals

- What about subtraction?
  - Easy once you have ‘predecessor’, but...
  - Predecessor is very difficult!

- Story time:
  - One of Church’s students, Kleene (of Kleene-star fame) was struggling to think of how to encode ‘predecessor’, until it came to him during a trip to the dentists office.
  - Take from this what you will

- Wikipedia has a great derivation of ‘predecessor’, not enough time today.
Looping + Recursion

- So far we have avoided self-reference, so how does recursion work?
- We can construct a lambda term that ‘replicates’ itself:
  - Define $D = \lambda x.x x$, then
    - $D D = (\lambda x.x x) (\lambda x.x x) \rightarrow (\lambda x.x x) (\lambda x.x x) = D D$
  - $D D$ is an infinite loop
- We want to generalize this, so that we can make use of looping
The Fixpoint Combinator

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

- Then
  \(Y F =\)
  \[(\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) F \rightarrow\]
  \[(\lambda x. F (x x)) (\lambda x. F (x x)) \rightarrow\]
  \(F ((\lambda x. F (x x)) (\lambda x. F (x x)))\)
  \(= F (Y F)\)

- \(Y F\) is a fixed point (aka fixpoint) of \(F\)
- Thus \(Y F = F (Y F) = F (F (Y F)) = \ldots\)
  - We can use \(Y\) to achieve recursion for \(F\)
Example

\[ \text{fact} = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times (f (n-1)) \]

• The second argument to \text{fact} is the integer
• The first argument is the function to call in the body
  ➢ We’ll use \text{Y} to make this recursively call \text{fact}

\[(\text{Y fact}) 1 = \text{fact (Y fact)) 1}\]

\[\rightarrow \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 \times ((\text{Y fact}) 0)\]

\[\rightarrow 1 \times ((\text{Y fact}) 0)\]

\[= 1 \times (\text{fact (Y fact) 0})\]

\[\rightarrow 1 \times (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 \times ((\text{Y fact}) (-1)))\]

\[\rightarrow 1 \times 1 \rightarrow 1\]
Factorial 4=

(Y G) 4

G (Y G) 4

(\lambda r. \lambda n. (if n = 0 then 1 else n \times (r (n-1)))) (Y G) 4

(\lambda n. (if n = 0 then 1 else n \times ((Y G) (n-1)))) 4

if 4 = 0 then 1 else 4 \times ((Y G) (4-1))

4 \times (G (Y G) (4-1))

4 \times ((\lambda n. (1, if n = 0; else n \times ((Y G) (n-1)))) (4-1))

4 \times (1, if 3 = 0; else 3 \times ((Y G) (3-1)))

4 \times (3 \times (G (Y G) (3-1)))

4 \times (3 \times ((\lambda n. (1, if n = 0; else n \times ((Y G) (n-1)))) (3-1)))

4 \times (3 \times (1, if 2 = 0; else 2 \times ((Y G) (2-1))))

4 \times (3 \times (2 \times (G (Y G) (2-1))))

4 \times (3 \times (2 \times ((\lambda n. (1, if n = 0; else n \times ((Y G) (n-1)))) (2-1))))

4 \times (3 \times (2 \times (1, if 1 = 0; else 1 \times ((Y G) (1-1)))))

4 \times (3 \times (2 \times (1 \times (G (Y G) (1-1)))))

4 \times (3 \times (2 \times (1 \times ((\lambda n. (1, if n = 0; else n \times ((Y G) (n-1)))) (1-1)))))

4 \times (3 \times (2 \times (1 \times (1, if 0 = 0; else 0 \times ((Y G) (0-1)))))

4 \times (3 \times (2 \times (1 \times (1))))

24
Discussion

- Lambda calculus is Turing-complete
  - Most powerful language possible
  - Can represent pretty much anything in “real” language
    - Using clever encodings
- But programs would be
  - Pretty slow (10000 + 1 → thousands of function calls)
  - Pretty large (10000 + 1 → hundreds of lines of code)
  - Pretty hard to understand (recognize 10000 vs. 9999)
- In practice
  - We use richer, more expressive languages
  - That include built-in primitives
The Need For Types

- Consider the **untyped** lambda calculus
  - \( \text{false} = \lambda x.\lambda y.y \)
  - \( 0 = \lambda x.\lambda y.y \)

- Since everything is encoded as a function...
  - We can easily misuse terms...
    - \( \text{false} 0 \rightarrow \lambda y.y \)
    - \( \text{if } 0 \text{ then } ... \)

...because everything evaluates to some function

- The same thing happens in assembly language
  - Everything is a machine word (a bunch of bits)
  - All operations take machine words to machine words
Simply-Typed Lambda Calculus (STLC)

- $e ::= n \mid x \mid \lambda x: t. e \mid e \ e$
  - Added integers $n$ as primitives
    - Need at least two distinct types (integer & function)…
    - …to have type errors
  - Functions now include the type $t$ of their argument

- $t ::= \text{int} \mid t \rightarrow t$
  - int is the type of integers
  - $t_1 \rightarrow t_2$ is the type of a function
    - That takes arguments of type $t_1$ and returns result of type $t_2$
Types are limiting

- STLC will reject some terms as ill-typed, even if they will not produce a run-time error
  - Cannot type check Y in STLC
    - Or in OCaml, for that matter, at least not as written earlier.
  - Surprising theorem: All (well typed) simply-typed lambda calculus terms are strongly normalizing
    - A normal form is one that cannot be reduced further
      - A value is a kind of normal form
    - Strong normalization means STLC terms always terminate
      - Proof is not by straightforward induction: Applications “increase” term size
Summary

- Lambda calculus is a core model of computation
  - We can encode familiar language constructs using only functions
    - These encodings are enlightening – make you a better (functional) programmer

- Useful for understanding how languages work
  - Ideas of types, evaluation order, termination, proof systems, etc. can be developed in lambda calculus,
    - then scaled to full languages