**Lambda Calculus**

- Proposed in 1930s by
  - Alonzo Church
  - Stephen Cole Kleene

- 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...

**Lambda Calculus (λ-calculus)**

A lambda calculus expression is defined as

\[ e ::= x \quad |\quad \lambda x.e \quad |\quad ee \]

- variable
- function
- function application

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

That’s it! Nothing but higher-order functions

**Programming Language Features**

- Many features exist simply for convenience
  - Multi-argument functions
    - Use currying or tuples
  - Loops
    - Use recursion
  - Side effects
    - Use functional programming

- So what language features are really needed?

**Turing Completeness**

- Computational system that can
  - Simulate a Turing machine
  - Compute every Turing-computable function

- 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 Theory**

- Come up with a “core” language
  - That’s as small as possible
  - But still Turing complete

- Helps illustrate important
  - Language features
  - Algorithms

- One solution
  - Lambda calculus

**Lambda Expressions**

- A lambda calculus expression is defined as

\[ e ::= x \quad |\quad \lambda x.e \quad |\quad ee \]

- variable
- function
- function application

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

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

- Syntactic sugar for local declarations
  - let x = e1 in e2 is short for (\x.e2) e1

- 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

---

Lambda Calculus Semantics

- All we’ve got are functions
  - So all we can do is call them

- To evaluate (\x.e1) e2
  - Evaluate e1 with x bound to e2

- This application is called beta-reduction
  - (\x.e1) e2 \( \rightarrow \) e1[x/e2]
    - e1[x/e2] is e1 where occurrences of x are replaced by e2
    - Slightly different than the environments we saw for OCaml
    - Do substitutions to replace formals with actuels
    - Instead of using environment to map formals to actuels
    - We allow reductions to occur anywhere in a term

---

Beta Reduction Example

\[
(\lambda x. (\lambda z. x) \ y) \\
\rightarrow (\lambda x. (\lambda z. (x) \ y)) \ y \quad \text{// since \( \lambda \) extends to right} \\
\rightarrow (\lambda x. (\lambda z. (x) \ y)) \ y \quad \text{// apply (\lambda x.e1) e2 \( \rightarrow \) e1[x/e2]} \\
\rightarrow \lambda z.(y \ z) \quad \text{// final result} \\
\]

- Equivalent OCaml code
  - (fun x -> (fun z -> (x y) z)) y \( \rightarrow \) fun z -> (y z)

---

Lambda Calculus Examples

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

- (\lambda x. \lambda y. x) z \( \rightarrow \) \lambda y. z y
  - A function that applies its argument to y

---

Lambda Calculus Examples (cont.)

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

---

Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping

- Consider the following
  - (\lambda x. (\lambda x. x)) z \( \rightarrow \) ?
    - 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. (\lambda y. y))
  - Renaming bound variables consistently is allowed
    - This is called alpha-renaming or alpha conversion
  - Ex. \( \lambda x. x = \lambda y. y \rightarrow \lambda z. z \) \( \lambda y. \lambda x. x \) \( \lambda z. \lambda x. z \)
Static Scoping (cont.)

- How about the following?
  - \((\lambda x. \lambda y. x) y\) \(\rightarrow\) ?
  - When we replace \(y\) inside, we don’t want it to be captured by the inner binding of \(y\)
  - I.e., \((\lambda x. \lambda y. x) y = \lambda y. y\)

- Solution
  - \((\lambda x. \lambda y. x) y\) is “the same” as \((\lambda x. \lambda z. x)z\)
  - Due to alpha conversion
  - So change \((\lambda x. \lambda y. x) y\) to \((\lambda x. \lambda z. x) z\) \(y\) first
    - Now \((\lambda x. \lambda z. x) z\) \(y\) \(\rightarrow\) \(\lambda z. y\) \(z\)

Encodings

- The lambda calculus is Turing complete

- Means we can encode any computation we want
  - If we’re sufficiently clever...

Examples

- Booleans
- Pairs
- Natural numbers & arithmetic
- Looping

Booleans (cont.)

- Other Boolean operations
  - \(\text{not} = \lambda x. ((x \text{ false}) \text{ true})\)
    - \(\text{not} \rightarrow (\lambda x. ((x \text{ false}) \text{ true}) \rightarrow (\text{true false}) \rightarrow \text{false}\)
  - \(\text{and} = \lambda x. \lambda y. ((xy) \text{ false})\)
  - \(\text{or} = \lambda x. \lambda y. ((x \text{ true}) y)\)

- Given these operations
  - Can build up a logical inference system

Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - \((\lambda x. e_1) e_2 \rightarrow e_1[x/e_2]\)
  - We must first alpha-convert variables as necessary
  - Usually performed implicitly (w/o showing conversion)

- Examples
  - \((\lambda x. \lambda y. x) y = (\lambda x. \lambda z. x) z \rightarrow \lambda z. y z\) \(\parallel y \rightarrow z\)
  - \((\lambda x. (\lambda x. x)) z = (\lambda y. (\lambda x. x)) z \rightarrow z (\lambda x. x)\) \(\parallel x \rightarrow y\)
  - \((\lambda x. (\lambda x. x)) z = (\lambda x. (\lambda y. y)) z \rightarrow z (\lambda y. y)\) \(\parallel x \rightarrow y\)

Booleans

- Church’s encoding of mathematical logic
  - \(\text{true} = \lambda x. \lambda y. x\)
  - \(\text{false} = \lambda x. \lambda y. y\)
  - if \(a\) then \(b\) else \(c\)
    - Defined to be the \(\lambda\) expression: \(a\) \(b\) \(c\)

- Examples
  - if true then \(b\) else \(c\) \(\rightarrow\) \((\lambda x. \lambda y. x) b c \rightarrow (\lambda y. b) c \rightarrow b\)
  - if false then \(b\) else \(c\) \(\rightarrow\) \((\lambda x. \lambda y. y) b c \rightarrow (\lambda y. y) c \rightarrow c\)

Pairs

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

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

- Encoding of non-negative integers
  - $0 = \lambda x. y$
  - $1 = \lambda x. y . f$
  - $2 = \lambda x. y . f (f y)$
  - $3 = \lambda x. y . f (f (f y))$
  - i.e., $n = \lambda x. y . \text{apply } f \text{ } n \text{ } \text{times to } y$

* (Alonzo Church, of course)

Operations On Church Numerals

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

- Example
  - $\text{succ} 0 =$
    - $(\lambda z. \lambda x. y . (z f y)) (\lambda x. y . y) \rightarrow$
    - $\lambda x. y . ((\lambda x. y . y) y) \rightarrow$
    - $\lambda x. y . \text{true} \rightarrow$ Since $(\lambda x. y . z \rightarrow y) \text{true}$

Operations On Church Numerals (cont.)

- IsZero?
  - $\text{iszero} = \lambda z. (\lambda y. \text{false}) \text{true}$
  - This is equivalent to $\lambda z. ((z (\lambda y. \text{false})) \text{true})$

- Example
  - $\text{iszero } 0 =$
    - $(\lambda z. (\lambda y. \text{false}) \text{true}) (\lambda x. y . y) \rightarrow$
    - $(\lambda x. y . y) \rightarrow$
    - $(\lambda y. \text{true} \rightarrow$ Since $(\lambda x. y . z \rightarrow y) \text{true}$

Arithmetic Using Church Numerals

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

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

- Multiplication
  - $M \times N = \lambda x . (M (N x))$
  - Equivalently: $\times = \lambda M . \lambda N . \lambda x . (M (N x))$
      - In prefix notation ($\times M N$)

Arithmetic (cont.)

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

- With these definitions
  - Can build a theory of arithmetic

Looping

- Define $D = \lambda x. x . x$, then
  - $D D = (\lambda x . x) (\lambda x . x) \rightarrow (\lambda x . x) (\lambda x . x) = D D$

- So $D D$ is an infinite loop
  - In general, self application is how we get looping
The “Paradoxical” Combinator

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

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

- Thus \( YF = F (YF) = F (F (YF)) = \ldots \)
  - 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 \( Y \) to make this recursively call \( \text{fact} \)

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

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 \rightarrow \text{thousands of function calls})
  - Pretty large (10000 + 1 \rightarrow \text{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

Simply-Typed Lambda Calculus

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

Simply-Typed Lambda Calculus (cont.)

- \( t ::= \text{int} \mid t ightarrow t \)
  - \( \text{int} \) is the type of integers
  - \( t \rightarrow t_2 \) is the type of a function
    - That takes arguments of type \( t_1 \) and returns result of type \( t_2 \)
  - \( t_1 \) is the domain and \( t_2 \) is the range
  - Notice this is a recursive definition
    - So we can give types to higher-order functions
- Will show how to compute types later
  - Example of operational semantics
Summary

- Lambda calculus shows issues with
  - Scoping
  - Higher-order functions
  - Types

- Useful for understanding how languages work