Lambda Calculus

Programming Language Features

- Many features exist simply 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

- 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 Calculus (λ-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 Expressions

- A lambda calculus expression is defined as

  e ::= x variable
  | λx.e function
  | e e function application

- λx.e is like (fun x -> 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 `(λx.e2) e1`

- Scope of `λ` extends as far right as possible
  - Subject to scope delimited by parentheses
  - `λx. λy.x y` is same as `λx. (λy. (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 `(λx.e1) e2`
  - Evaluate `e1` with `x` bound to `e2`

- This application is called beta-reduction
  - `(λx.e1) e2 → e1[e2/x]`
    - `e1[e2/x]` 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 actuals
      - Instead of using environment to map formals to actuals
  - We allow reductions to occur anywhere in a term

Lambda Calculus Examples

- `(λx.x z) z` → `z`
- `(λx.x) x` → `x`
- `(λx.x (λy.y)) z` → `z`

Lambda Calculus Examples (cont.)

- `(λx.(λz.x z)) y` → `(λz.x z)`
- `(λx.(λz.x z)) (λz.z)` → `(λz.z)`

Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping

- Consider the following
  - `(λx.(λy.x)) z` → ?
    - 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 `(λx.(λy.y))`
    - Renaming bound variables consistently is allowed
    - This is called alpha-renaming or alpha conversion
      - Ex. `λx.x = λy.y = λz.z`
      - `λy.λx.y = λz.λx.z`
Static Scoping (cont.)

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

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

Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - \((\lambda x.e1) e2\) \to \(e1[x/e2]\)
  - We must first alpha-convert variables as necessary
  - Usually performed implicitly (w/o showing conversion)

- Examples
  - \((\lambda x.\lambda y.x y)\) y = \(\lambda z.y z\)
  - \((\lambda x.(\lambda z.z)) z\) = \(\lambda x.(\lambda z.z)\) z \to z \(\lambda x.x\) \hspace{1cm} \(\lambda z.z\)
  - \((\lambda z.(\lambda x.(\lambda y.y))) z\) = \(\lambda z.(\lambda x.(\lambda y.y))) z\) z \(\lambda z.(\lambda y.y)\) \hspace{1cm} \(\lambda z.z\)

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

- Church’s encoding of mathematical logic
  - true = \(\lambda x.\lambda y.x\)
  - 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 = \((\lambda x.\lambda y.x)\) b c = \(\lambda y.b\) c = b
  - if false then b else c = \((\lambda x.\lambda y.y)\) b c = \(\lambda y.c\) c = c

Booleans (cont.)

- Other Boolean operations
  - not = \(\lambda x.((x\ false)\ true)\)
    - not true \to \(\lambda x.((x\ false)\ true)\) true \to \(\true\ false\ true\) \to \false
  - and = \(\lambda x.\lambda y.((xy)\ false)\)
  - or = \(\lambda x.\lambda y.((x\ true)\ y)\)

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

Pairs

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

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

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

*(Alonzo Church, of course)*

Operations On Church Numerals

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

- **Example**
  - \( \text{succ} \ 0 = \ 0 = \ 1 \)

Arithmetic Using Church Numerals

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

- **Addition**
  - \( M + N = \lambda x. \lambda y. \text{(+ )} (\lambda x. \lambda y. x) (\lambda x. \lambda y. y) \)
  
  Equivalently: \( + = \lambda M. \lambda N. \lambda x. \lambda y. \text{(+ )} (\lambda x. \lambda y. x) (\lambda x. \lambda y. y) \)
  
  In prefix notation: \( (+ M N) \)

- **Multiplication**
  - \( M \ast N = \lambda x. \lambda y. \text{(\times )} (\lambda x. \lambda y. x) (\lambda x. \lambda y. y) \)
  
  Equivalently: \( \ast = \lambda M. \lambda N. \lambda x. \lambda y. \text{(\times )} (\lambda x. \lambda y. x) (\lambda x. \lambda y. y) \)
  
  In prefix notation: \( (* M N) \)

Arithmetic (cont.)

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

- **With these definitions**
  - Can build a theory of arithmetic

Looping

- **Define D = \( \lambda x. x \), then**
  - \( D \ D = \ 0 = \ 1 \)

- **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

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

Thus Y F = F (Y F) = F (F (Y F)) = ...

• We can use Y to achieve recursion for F

Example

fact = \lambda f. \lambda n. if n = 0 then 1 else n * (f (n-1))

• The second argument to fact is the integer
• The first argument is the function to call in the body
  > We’ll use Y to make this recursively call fact

(Y fact) 1 = (fact (Y fact)) 1
  → if 1 = 0 then 1 else 1 * ((Y fact) 0)
  → 1 * ((Y fact) 0)
  → 1 * (fact (Y fact) 0)
  → 1 * (if 0 = 0 then 1 else 0 * ((Y fact) (-1))
  → 1 * 1 → 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 → 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
  > false = \lambda x.\lambda y. y
  > 0 = \lambda x.\lambda y. y
• Since everything is encoded as a function...
  > We can easily misuse terms...
    > false 0 \rightarrow \lambda y. y
    > if 0 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

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

Simply-Typed Lambda Calculus (cont.)

• t ::= int | t \rightarrow t
  > int is the type of integers
  > t1 \rightarrow t2 is the type of a function
    > That takes arguments of type t1 and returns result of type t2
  > t1 is the domain and t2 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