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?

Programming Language Theory
- Come up with a “core” language
  - That’s as small as possible
  - But still Turing complete
    > Can map every Turing machine to a program
- Helps illustrate important
  - Language features
  - Algorithms
- One solution
  - Lambda calculus

Lambda Calculus (\(\lambda\)-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 Expressions

A lambda calculus expression is defined as

\[ e ::= x \quad \text{variable} \]
\[ | \lambda x. e \quad \text{function} \]
\[ | e \; e \quad \text{function application} \]

- \( \lambda x. e \) is like \((\text{fun } x \rightarrow e)\) in OCaml
- That's it! Nothing but higher-order functions

Intuitive Understanding

Before we work more with the mathematical notation of lambda calculus, we're going to play a puzzle game!

From: http://worrydream.com/AlligatorEggs/

Puzzle Pieces

- Hungry alligators: eat and guard family
- Old alligators: guard family
- Eggs: hatch into new family
- Lambda abstraction (\(\lambda x\))
- Parentheses (scope)
- Variables (\(e ::= x\))

Example Families

Families are shown in columns
Alligators guard families below them
Puzzle Rule 1: The Eating Rule

- If families are side-by-side the top left alligator eats the entire family to her right.
- The top left alligator dies.
- Any eggs she was guarding of the same color hatch into what she just ate.
- Newborn hungry alligators will then eat any eggs directly in front of them.
  - Follow same eating rule.

Eating Rule Practice

What happens to these alligators?

Puzzle 1: [Image]

Puzzle 2: [Image]

Eating rule practice

[Image]

Eating Rule Practice

What happens to these alligators?

Puzzle 1: [Image]

Puzzle 2: [Image]

Answer 1: [Image]

Answer 2: [Image]
Puzzle Rule 2: The Color Rule

- If an alligator is about to eat a family and a color appears in both families then we need to change that color in one of the families.

- If a color appears in both families, but only as an egg, no color change is made.

Puzzle Rule 3: The Old Alligator Rule

- When an old alligator is only guarding one family it dies.

Challenging Puzzles!

- Try to reduce these groups of alligators as much as possible using the three puzzle rules:

More Puzzles

- When Family Not eats Family True it becomes Family False and when Not eats False it becomes True… what color should the white eggs be?
Lambda Calculus

A lambda calculus expression is defined as

\[ e ::= x \quad \text{variable} \quad (x: \text{egg}) \]
\[ \lambda x.e \quad \text{function} \quad (\lambda x: \text{alligator}) \]
\[ e e \quad \text{function application} \quad (\text{adjacency of families}) \]

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

That’s it! Only higher-order functions

---

Three Conveniences

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

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

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

This is the eating rule!
Beta Reduction Example

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

\[\rightarrow (\lambda x. (\lambda z. (x \ z))) \ y\] // since \(\lambda\) extends to right

\[\rightarrow (\lambda z. (\lambda z. (\lambda z. (x \ z)))) \ y\] // apply \((\lambda x. e_1) \ e_2 \rightarrow e_1[x/e_2]\)

// where \(e_1 = \lambda z. (x \ z)\), \(e_2 = y\)

\[\rightarrow \lambda z. (y \ z)\] // final result

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

Lambda Calculus Examples

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

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

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

- A function that applies its argument to \(y\)

Parameters
- Formal
- Actual

Lambda Calculus Examples (cont.)

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

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

- A curried function of two arguments
- Applies its first argument to its second

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

Try these with alligators, too!

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. \(\lambda x. x = \lambda y. y = \lambda z. z\)
  - \(\lambda y. \lambda z. y = \lambda z. \lambda x. z\)
Static Scoping (cont.)

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 \(y\)
  • I.e., \((\lambda x. \lambda y. x y) y \neq \lambda y. y y\)

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

Beta-Reduction, Again

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

Examples
• \((\lambda x. \lambda y. x y) y = (\lambda x. \lambda z. x z) y \rightarrow \lambda z. y z\)  // \(y \rightarrow z\)
• \((\lambda x. (\lambda x). x) z = (\lambda y. (\lambda x). x) z \rightarrow z (\lambda x)\)  // \(x \rightarrow y\)
• \((\lambda x. (\lambda x). x) z = (\lambda x. (\lambda y. y)) z \rightarrow z (\lambda y. y)\)  // \(x \rightarrow y\)

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 \(c\) \(\rightarrow (\lambda x. \lambda y. x) \ b\ c \rightarrow (\lambda y. b)\ c \rightarrow b\)
• if \(false\) then \(b\) else \(c\) \(\rightarrow (\lambda x. \lambda y. y) \ b\ c \rightarrow (\lambda y. y)\ c \rightarrow c\)
Booleans (cont.)

- Other Boolean operations
  - `not = λx.((x false) true)`
  - `not true → λx.((x false) true) true → (true false) → false`
  - `and = λx.λy.((xy) false)`
  - `or = λx.λy.((x true) y)`

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

Pairs

- Encoding of a pair `a, b`
  - `(a,b) = λx.if x then a else b`
  - `fst = λf.f true`
  - `snd = λf.f false`

- Examples
  - `fst (a,b) = (λf.f true)(λx.if x then a else b) → (λx.if x then a else b) true → if true then a else b → a`
  - `snd (a,b) = (λf.f false)(λx.if x then a else b) → (λx.if x then a else b) false → if false then a else b → b`

Natural Numbers (Church* Numerals)

- Encoding of non-negative integers
  - `0 = λf.λy.y`
  - `1 = λf.λy.f y`
  - `2 = λf.λy.f (f y)`
  - `3 = λf.λy.f (f (f y))`
  - i.e., `n = λf.λy.<apply f n times to y>`

* (Alonzo Church, of course)

Operations On Church Numerals

- Successor
  - `succ = λz.λf.λy.f (z f y)`
  - `0 = λf.λy.y`
  - `1 = λf.λy.f y`

- Example
  - `succ 0 = (λz.λf.λy.f (z f y))(λf.λy.y) → λf.λy.((λf.λy.y) f y) → λf.λy.f ((λf.λy.y) y) → λf.λy.f y y`  \[ \text{Since } (λx.y) z → y \]
  - `λf.λy.f y`  \[ = 1 \]
Operations On Church Numerals (cont.)

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

- Example
  - iszero 0 =
    (λz.(λy.false) true) (λf.λy.y)
    (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 = λx.λy.(M x)((N x) y)
    Equivalently: + = λM.λN.λx.λy.(M x)((N x) y)
    - In prefix notation (+ M N)

  - Multiplication
    - M * N = λx.λy.(M x)(N y)
    Equivalently: * = λM.λN.λx.λy.(M x)(N y)
    - In prefix notation (* M N)

Arithmetic (cont.)

- Prove 1+1 = 2
  - 1+1 = λx.λy.(1 x)((1 x) y) =
    λx.λy.(λx.λy.x y)((λx.λy.x y) x) y) →
    λx.λy.(λx.λy.x y)(λx.λy.x y) y) →
    λx.λy.(λx x y)(λx x y) y) →
    λx.λy.x (λy.y) y) →
    λx.λy.(x y) = 2
  - Many implicit alpha conversions

- With these definitions
  - Can build a theory of arithmetic

Looping

- Define D = λx.x x, then
  - D D = (λx.x x) (λx.x x) → (λx.x x) (λx.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

\[ Y F = \]

\[ (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) F \]

\[ = (\lambda x. F (x x)) (\lambda x. F (x x)) \]

\[ = F (Y F) \]

Thus \( Y F = F (Y F) = F (F (Y F)) = \ldots \)

• We can use \( Y \) to achieve recursion for \( F \)

Example

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

• 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 \( \text{fact} \)

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

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

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

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

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

\[ \rightarrow 1 \ast 1 \rightarrow 1 \]

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 thousands of function calls)
  • Pretty large (10000 + 1 \rightarrow 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

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

Simply-Typed Lambda Calculus (cont.)

- $t ::= \text{int} \mid t \rightarrow t$
  - 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