Lambda Calculus

- 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 \((\text{fun } x \to e)\) in OCaml

- That’s it! Only higher-order functions
Beta-Reduction, Again

• Whenever we do a step of beta reduction...
  – \((\lambda x.e_1) e_2 \rightarrow e_1[x/e_2]\)
  – ...alpha-convert variables as necessary

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

Encodings

• It turns out that this language is Turing complete

• That means we can encode any computation we want in it
  – ...if we’re sufficiently clever...
Booleans

The lambda calculus was created by logician Alonzo Church in the 1930's to formulate a mathematical logical system

\[ \text{true} = \lambda x.\lambda y.x \]
\[ \text{false} = \lambda x.\lambda y.y \]

if a then b else c is 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 (continued)

Other Boolean operations:
• not = \( \lambda x.((x \ false) \ true) \)
• not true \( \rightarrow \lambda x.((x \ false) \ true) \ true \rightarrow ((true \ false) \ true) \rightarrow false \)
• and = \( \lambda x.\lambda y.((xy) \ false) \)
• or = \( \lambda x.\lambda y.((x \ true) \ y) \)
• Show not, and and or have the desired properties, …
• Given these operations, can build up a logical inference system
Pairs

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

• Examples:
  – 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) \text{ true } \rightarrow
    \text{ if true } \text{ then } a \text{ else } b \rightarrow a
  – 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) \text{ false } \rightarrow
    \text{ if false } \text{ then } a \text{ else } b \rightarrow b

Natural Numbers (Church*)

*(Named after Alonzo Church, developer of lambda calculus)

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 n times to y>}

succ = \lambda z. \lambda f. \lambda y. f \ (z \ f \ y)
iszero = \lambda g. g \ (\lambda y. \text{false}) \text{ true }
  – Recall that this is equivalent to \lambda g. ((g \ (\lambda y. \text{false})) \text{ true})
Natural Numbers (cont’d)

• Examples:
  succ 0 =
  \((\lambda z.\lambda f.\lambda y. f \,(z \,f \,y)) \,(\lambda f.\lambda y. y) \rightarrow\) 
  \(\lambda f.\lambda y. f \,((\lambda f.\lambda y. y) \,f \,y) \rightarrow\) 
  \(\lambda f.\lambda y. y \,= \,1\)

  iszero 0 =
  \((\lambda z.\,z \,(\lambda y.\,\text{false}) \,\text{true}) \,(\lambda f.\lambda y. y) \rightarrow\) 
  \((\lambda f.\lambda y. y) \,(\lambda y.\,\text{false}) \,\text{true} \rightarrow\) 
  \((\lambda y. y) \,\text{true} \rightarrow\) 
  \text{true}

Arithmetic defined

• Addition, if \(M\) and \(N\) are integers (as \(\lambda\) expressions):
  \[ M + N = \lambda x.\lambda y. \,(M \,x)((N \,x) \,y) \]
  Equivalently:
  \[ + = \lambda M.\lambda N.\lambda x.\lambda y. \,(M \,x)((N \,x) \,y) \]
• Multiplication:
  \[ M \,* \,N = \lambda x. \,(M \,(N \,x)) \]
• Prove \(1+1 = 2\).
  \[ 1+1 = \lambda x.\lambda y. \,(1 \,x)(((1 \,x) \,y) \rightarrow\) 
  \[ \lambda x.\lambda y. \,(\lambda x.\lambda y. \,x \,y)(((\lambda x.\lambda y. \,x \,y) \,y) \rightarrow\) 
  \[ \lambda x.\lambda y. \,(\lambda y. \,x \,y)(((\lambda y. \,x \,y) \,y) \rightarrow\) 
  \[ \lambda x.\lambda y. \,((\lambda y. \,x \,y) \,y) \rightarrow\] 
  \[ \lambda x.\lambda y. \,(x \,y) \,= \,2\]
• With these definitions, can build a theory of integer arithmetic.
Looping

- Define $D = \lambda x.x x$
- Then
  - $D D = (\lambda x.x x) (\lambda x.x x) \rightarrow (\lambda x.x x) (\lambda x.x x) = D D$
- 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 $Y F = F (Y F) = F (F (Y F)) = ...$
Example

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

- 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}

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

Discussion

- Using encodings we can represent pretty much anything we have in a “real” language
  - But programs would be pretty slow if we really implemented things this way
  - In practice, we use richer languages that include built-in primitives

- Lambda calculus shows all the issues with scoping and higher-order functions

- It's useful for understanding how languages work
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 ...
      • 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

What is a Type System?

• A *type system* is some mechanism for distinguishing good programs from bad
  – Good = well typed
  – Bad = ill typed or not typable; has a *type error*

• Examples
  – 0 + 1 // well typed
  – false 0 // ill-typed; can't apply a boolean
Static versus Dynamic Typing

- In a *static type system*, we guarantee at compile time that all program executions will be free of type errors
  - OCaml and C have static type systems

- In a *dynamic type system*, we wait until runtime, and halt a program (or raise an exception) if we detect a type error
  - Ruby has a dynamic type system

- Java, C++ have a combination of the two

Simply-Typed Lambda Calculus

- e ::= n | x | λx:t.e | e e
  - We’ve added integers n as primitives
    - Without at least two distinct types (integer and function), can’t have any type errors
  - Functions now include the type of their argument

- t ::= int | t → t
  - int is the type of integers
  - t1 → t2 is the type of a function that takes arguments of type t1 and returns a result of type t2
  - t1 is the *domain* and t2 is the *range*
  - Notice this is a recursive definition, so that we can give types to higher-order functions
Type Judgments

- We will construct a type system that proves judgments of the form

\[ A \vdash e : t \]

- “In type environment A, expression e has type t”

- If for a program e we can prove that it has some type, then the program type checks
  - Otherwise the program has a type error, and we’ll reject the program as bad

Type Environments

- A type environment is a map from variables names to their types
  - Just like in our operational semantics for Scheme

- is the empty type environment

- A, x:t is just like A, except x now has type t

- When we see a variable in the program, we’ll look up its type in the environment
Type Rules

e ::= n | x | λx:t.e | e e

\[ \frac{\text{A} \vdash n : \text{int}}{\text{A} \vdash \text{x} \in \text{A}} \]

\[ \frac{\text{A, x : t} \vdash e : t'}{\text{A} \vdash \lambda x : t.e : t \rightarrow t'} \]

\[ \frac{\text{A} \vdash e : t \rightarrow t' \quad \text{A} \vdash e' : t}{\text{A} \vdash e e' : t'} \]

Example

\[ A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]
\[ B = A, x : \text{int} \]

\[ \frac{B \vdash + : i \rightarrow i \rightarrow i}{B \vdash x : \text{int}} \]

\[ \frac{B \vdash + x : \text{int} \rightarrow \text{int}}{B \vdash 3 : \text{int}} \]

\[ \frac{B \vdash + 3 : \text{int}}{A \vdash (\lambda x : \text{int}. + x 3) : \text{int} \rightarrow \text{int}} \]

\[ A \vdash 4 : \text{int} \]

\[ A \vdash (\lambda x : \text{int}. + x 3) 4 : \text{int} \]
Discussion

- The type rules are a kind of logic for reasoning about types of programs
  - The tree of judgments we just saw is a kind of proof in this logic that the program has a valid type

- So the type checking problem is like solving a jigsaw puzzle
  - Can we apply the rules to a program in such a way as to produce a typing proof?
  - It turns out we can easily decide whether or not we can do this.

An Algorithm for Type Checking

(Write this in OCaml!)

TypeCheck : type env × expression → type

TypeCheck(A, n) = int
TypeCheck(A, x) = if x in A then A(x) else fail
TypeCheck(A, λx:t.e) =
  let t' = TypeCheck((A, x:t), e) in t → t'
TypeCheck(A, e1 e2) =
  let t1 = TypeCheck(A, e1) in
  let t2 = TypeCheck(A, e2) in
  if dom(t1) = t2 then range(t1) else fail
**Type Inference**

- We could extend the rules to show how a language could figure out, even if types aren't specified, what the types of everything are in a program
  - Can you believe there are languages which can actually do this?
- We could do these things, but we actually won't.

**Summary**

- Lambda calculus shows all the issues with scoping and higher-order functions
- It's useful for understanding how languages work
Practice

• Reduce the following:
  – $(\lambda x. \lambda y. x \ y) \ (\lambda a. a) \ b$
  – $(\text{or true}) \ (\text{and true false})$
  – $(^* \ 1 \ 2) \quad (^* \ m \ n = \lambda M. \lambda N. \lambda x. (M \ (N \ x)) )$

• Derive and prove the type of:
  – $(\lambda f: \text{int}\to \text{int}. \lambda n: \text{int}. \ f \ n) \ (\lambda x: \text{int}. \ 3 + x) \ 6$
  – $\lambda x: \text{int}\to\text{int}. \lambda y: \text{int}\to\text{int}. \lambda z: \text{int}. \ x \ (y \ z)$