Lambda Calculus

A lambda calculus expression is defined as

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

- \( \lambda x. e \) is like `(fun x -> e)` in OCaml

- That's it! Only higher-order functions

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

- \( \text{true} = \lambda x. \lambda y. x \)
- \( \text{false} = \lambda x. \lambda y. y \)
- \( \text{if } a \text{ then } b \text{ else } c \) 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:

- \( \text{not} = \lambda x.(x \; \text{false}) \)
- \( \text{not true} \rightarrow \lambda x.(x \; \text{false}) \; \text{true} \rightarrow (\text{true false}) \; \text{true} \rightarrow \text{false} \)
- \( \text{and} = \lambda x.\lambda y.(x y) \; \text{false} \)
- \( \text{or} = \lambda x.\lambda y.(x \; \text{true}) \; y \)

- Show not, and or have the desired properties, ...

- Given these operations, can build up a logical inference system
Pairs

\[(a,b) = \lambda x. \text{if } x \text{ then } a \text{ else } b\]

\[\text{fst} = \lambda f. \text{true}\]

\[\text{snd} = \lambda f. \text{false}\]

- **Examples:**
  - \[\text{fst} (a,b) = (\lambda f. \text{true}) (\lambda x. \text{if } x \text{ then } a \text{ else } b)\]
  - \[\text{snd} (a,b) = (\lambda f. \text{false}) (\lambda x. \text{if } x \text{ then } a \text{ else } b)\]

Natural Numbers (Church*)

*Named after Alonzo Church, developer of lambda calculus*

\[0 = \lambda f. y. y\]

\[1 = \lambda f. y. f y\]

\[2 = \lambda f. y. f (f y)\]

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

\[\text{suc} = \lambda z. M. L y. f (z f y)\]

\[\text{iszero} = \lambda g. (\lambda y. \text{false}) \text{ true}\]

- Recall that this is equivalent to \[\lambda g. ((\lambda y. \text{false}) \text{ true})\]

Arithmetic defined

- Addition, if \(M\) and \(N\) are integers (as \(\lambda\) expressions):
  \[M + N = \lambda x. \lambda y. (M x)((N x) y)\]
  Equivalently: \[+ = \lambda M. \lambda N. \lambda x. (M x)((N x) y)\]

- Multiplication:
  \[M * N = \lambda x. (M (N x))\]

- Prove \(1+1 = 2\).
  \[1+1 = \lambda x. \lambda y. (\text{false}) ((\lambda x y. \text{true}) \text{ true})\]

- With these definitions, can build a theory of integer arithmetic.

Looping

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

- Thus \[Y F = F (Y F) = F ((Y F) F) = \ldots\]
Example

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

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

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

\[ e ::= \text{n | x | } \lambda x : t . e | e \]
- 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

\[ t ::= \text{int | } t \rightarrow t \]
- \( \text{int} \) 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 a result of type \( t_2 \)
- \( t_1 \) is the domain and \( t_2 \) 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
- \( \emptyset \) 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

\[ e ::= n | x | \lambda x.t.e | e \ e \]

\[
\begin{align*}
A \vdash n &: \text{int} & & x \cdot A & & A \vdash x &: A(x) \\
A, x : t \vdash e &: t' & & A, e &: t \vdash e' &: t' & & A \vdash \lambda x.t.e &: t \rightarrow t' \\
A, e &: t \rightarrow t' & & A \vdash e &: t' \\
A \vdash e \ e' &: t'
\end{align*}
\]

Example

\[
\begin{align*}
A &= \ + : \text{int} \rightarrow \text{int} & B &= A, x : \text{int} \\
B &= A, x : \text{int} & B &= \ 3 : \text{int} \\
B &= \ 3 : \text{int} & B &= \ x : \text{int} \\
B &= A, \ (\lambda x.\text{int}.\ x \ 3) : \text{int} \rightarrow \text{int} & A &= 4 : \text{int} \\
A &= (\lambda x.\text{int}.\ x \ 3) \ 4 : \text{int}
\end{align*}
\]

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!)

\[
\begin{align*}
\text{TypeCheck}(A, n) &= \text{int} \\
\text{TypeCheck}(A, x) &= \text{if } x \text{ in A then } A(x) \text{ else fail} \\
\text{TypeCheck}(A, \lambda x.t.e) &= \\
&\quad \text{let } t' = \text{TypeCheck}(A, x.t), e \text{ in } t \rightarrow t' \\
\text{TypeCheck}(A, e1 \ e2) &= \\
&\quad \text{let } t1 = \text{TypeCheck}(A, e1) \text{ in} \\
&\quad \text{let } t2 = \text{TypeCheck}(A, e2) \text{ in} \\
&\quad \text{if } \text{dom}(t1) = t2 \text{ then range}(t1) \text{ else fail}
\end{align*}
\]
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:
  - \((\lambda x. \lambda y. x y) (\lambda a. a) b\)
  - \((\text{true}) \ (\text{and} \ \text{true} \ \text{false})\)
  - \((\ast \ 1 \ 2)\) \((\ast \ m \ n = \lambda M. \lambda N. \lambda x. (M \ (N \ x))\))

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