CMSC 330: Organization of Programming Languages

Lambda Calculus and Types
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 \( \text{fun } x \rightarrow e \) 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:
  – $(\lambda x. x (\lambda x. x)) z = (\lambda x. x (\lambda y. y)) z \rightarrow z (\lambda y. y)$
  – $(\lambda x. \lambda y. x y) y = (\lambda x. \lambda z. x z) y \rightarrow \lambda z. y z$
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**

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

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

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

- Exercise: Show that not, and and or have the desired properties
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 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 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., \[ n = \lambda f. \lambda y. \langle \text{apply } f \ n \ \text{times to } y \rangle \]

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

• Examples:

\[ \text{succ 0} = \]
\[ (\lambda z.\lambda f.\lambda y. f (z f y)) (\lambda f.\lambda y. y) \rightarrow \]
\[ \lambda f.\lambda y. f ((\lambda f.\lambda y. y) f y) \rightarrow \]
\[ \lambda f.\lambda y. f y = 1 \]

\[ \text{iszero 0} = \]
\[ (\lambda z. z (\lambda y. \text{false}) \text{true}) (\lambda f.\lambda y. y) \rightarrow \]
\[ (\lambda f.\lambda y. y) (\lambda y. \text{false}) \text{true} \rightarrow \]
\[ (\lambda y. y) \text{true} \rightarrow \]
\[ \text{true} \]
**Arithmetic defined**

- Addition, if M and N are integers (as λ 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.
  \[ 1+1 = \lambda x.\lambda y.((1 x)((1 x) y)) \rightarrow \]
  \[ \lambda x.\lambda y.(((\lambda x.\lambda y.x y) x)(((\lambda x.\lambda y.x y) x) y)) \rightarrow \]
  \[ \lambda x.\lambda y.((\lambda y.x y)(((\lambda x.\lambda y.x y) x) y)) \rightarrow \]
  \[ \lambda x.\lambda y.(\lambda y.x y)(((\lambda x.\lambda y.x y) y) y) \rightarrow \]
  \[ \lambda x.\lambda y.((\lambda x.\lambda y.x y) y) \rightarrow \]
  \[ \lambda x.\lambda y.((\lambda y.x y) y) \rightarrow \]
  \[ \lambda x.\lambda y.x ((\lambda y.x y) y) \rightarrow \]
  \[ \lambda x.\lambda y.x (x y) = 2 \]
- 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

\[ Y \, F = \]

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

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

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

\[ = F (Y \, F) \]

• Thus \[ Y \, F = F (Y \, F) = F (F (Y \, F)) = \ldots \]
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 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

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

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

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

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

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

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

• 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 = λx.λy.y
  - 0 = λx.λy.y

- Since everything is encoded as a function...
  - We can easily misuse terms
    - false 0 → λy.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

- $e ::= n \mid x \mid \lambda x:t.e \mid e\ e$
  - 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

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

• \( A, x:t \) 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

\[ e ::= n \mid x \mid \lambda x:t.e \mid e \, e \]

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

\[ \frac{}{\frac{\text{TVar}}{A \vdash x : t}} \]

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

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

\[ A = \{ \text{+ : int} \to \text{int} \to \text{int} \} \]

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

\[ B \vdash \text{+ : i\to i\to i} \quad B \vdash x : \text{int} \]

\[ \begin{align*}
B \vdash \text{+ : i\to i\to i} & \quad B \vdash 3 : \text{int} \\
\hline
B \vdash \text{+ : i\to i\to i} & \quad B \vdash 3 : \text{int} \\
\hline
B \vdash \text{+ : i\to i\to i} & \quad B \vdash x : \text{int} \\
\hline
B \vdash \text{+ : i\to i\to i} & \quad B \vdash 3 : \text{int} \\
\hline
B \vdash \text{+ : i\to i\to i} & \quad B \vdash x : \text{int} \\
\hline
A \vdash (\lambda x:\text{int.}+: x 3) : \text{int} \to \text{int} & \quad A \vdash 4 : \text{int} \\
\hline
A \vdash (\lambda x:\text{int.}+: x 3) 4 : \text{int} 
\end{align*} \]
Discussion

• The type rules provide a way to reason about programs (i.e. a formal logic)
  – 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?
  – We can do this automatically
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 allow the type checker to deduce the types of every expression in a program even without the annotations
  – This is called type inference
  – Not covered in this class
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:
  – \((\lambda x.\lambda y.x \ y \ y) \ (\lambda a.a) \ b\)
  – \((\text{or true}) \ (\text{and true false})\)
  – \((\ast \ 1 \ 2)\)  \((\ast \ m \ n = \lambda M.\lambda N.\lambda x.(M \ (N \ x))\) \)

• Derive and prove the type of:
  – \(A \vdash (\lambda f:\text{int-}\to\text{int}.\lambda n:\text{int}.f \ n) \ (\lambda x:\text{int}.\ + \ 3 \ x) \ 6\)
    \[
    A = \{ + : \text{int} \to \text{int} \to \text{int} \}
    \]
  – \(\lambda x:\text{int-}\to\text{int-}\to\text{int}. \lambda y:\text{int-}\to\text{int}. \lambda z:\text{int}.x \ z \ (y \ z)\)