CMSC 330: Organization of Programming Languages

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
Turing Machine

Infinite Tape

1 0 0 0 1 1 1 1 0

Control Unit
State: Y

Read / Write Head

START

HALT

b; b, R
b; b, R
a; a, R
a; a, R

3

4

START

b; b, R
b; b, R
a; a, R
a; a, R

2

e; e, R
b; b, R
a; a, R
a; a, R

HALT
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
You only have:
• If statement
• Plus 1
• Minus 1

Sum $n = 1+2+3+4+5\ldots n$ in Mini C

```c
int add1(int n){return n+1;}
int sub1(int n){return n-1;}
int add(int a, int b){
    if(b == 0) return a;
    else return add(add1(a), sub1(b));
}
int sum(int n){
    if(n == 1) return 1;
    else return add(n, sum(sub1(n)));
}
int main(){
    printf("%d\n", sum(5));
}
```
Lambda Calculus (\(\lambda\)-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washington DC!)

- 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

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

- Note that this is CFG is ambiguous, but that’s not a problem for defining the terms in the language – we are not using it for parsing (i.e., different parse trees = different expressions)

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

- That’s it! Nothing but higher-order functions
Three Conveniences

- “Syntactic sugar” for local declarations
  - let \( x = e_1 \) in \( e_2 \) is short for \( (\lambda x. e_2) e_1 \)

- 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
OCaml implementation

```ocaml
type id = string

type exp = Var of id
| Lam of id * exp
| App of exp * exp

ey
λx.x
λx.λy.x y
(λx.λy.x y) λx.x x
```

Var "y"
Lam ("x", Var "x")
Lam ("x", (Lam ("y", App (Var "x", Var "y"))))
App (Lam ("x", Lam ("y", App (Var "x", Var "y")))))
Lam ("x", App (Var "x", Var "x")))
Lambda Calculus Semantics

- All we’ve got are functions
  - So all we can do is call them
- To evaluate \((\lambda x. e_1) \; e_2\)
  - Evaluate \(e_1\) with \(x\) replaced by \(e_2\)
- This application is called beta-reduction
  - \((\lambda x. e_1) \; e_2 \rightarrow e_1[x:=e_2]\)
    - \(e_1[x:=e_2]\) is \(e_1\) with occurrences of \(x\) replaced by \(e_2\)
    - This operation is called substitution
      - Replace formals with actuals
        - Instead of using environment to map formals to actuals
      - We allow reductions to occur anywhere in a term
        - Order reductions are applied does not affect final value!
Beta Reduction Example

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

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

\[\rightarrow (\lambda x.(\lambda z.(x z)))\ y\] // apply \((\lambda x.e1)\ e2\ \rightarrow e1[x:=e2]\ // where \(e1 = \lambda z.(x z)\), \(e2 = y\)

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

- Equivalent OCaml code
  - \((\text{fun } x \rightarrow (\text{fun } z \rightarrow (x z)))\ y \rightarrow \text{fun } z \rightarrow (y z)\)
Lambda Calculus Examples

- \((\lambda x.x) \ z \rightarrow z\)
- \((\lambda x.y) \ z \rightarrow y\)
- \((\lambda x.x \ y) \ z \rightarrow z \ y\)
  - A function that applies its argument to \(y\)
Lambda Calculus Examples (cont.)

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

- $(\lambda x.\lambda y.x \ y) \ z \rightarrow \lambda y.z \ y$
  - A curried function of two arguments
  - Applies its first argument to its second

- $(\lambda x.\lambda y.x \ y) \ (\lambda z.zz) \ x \rightarrow (\lambda y.(\lambda z.zz)y) \ x \rightarrow (\lambda z.zz) \ x \rightarrow xx$
Static Scoping & Alpha Conversion

- Lambda calculus uses **static scoping**

- Consider the following
  - \((\lambda x. x (\lambda x. x)) z \rightarrow ?\)
    - 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 \((\lambda x. x (\lambda y. y))\)
  - Renaming bound variables consistently is allowed
    - This is called **alpha-renaming** or **alpha conversion**
  - Ex. \(\lambda x. x = \lambda y. y = \lambda z. z\) \(\lambda y. \lambda x. y = \lambda z. \lambda x. z\)
Defining Substitution

- Use recursion on structure of terms
  - \( x[x:=e] = e \)  // Replace x by e
  - \( y[x:=e] = y \)  // y is different than x, so no effect
  - \( (e1 \ e2)[x:=e] = (e1[x:=e]) \ (e2[x:=e]) \)  // Substitute both parts of application
  - \( (\lambda x.e')[x:=e] = \lambda x.e' \)
    - In \( \lambda x.e' \), the x is a parameter, and thus a local variable that is different from other x’s. Implements static scoping.
    - So the substitution has no effect in this case, since the x being substituted for is different from the parameter x that is in e’
  - \( (\lambda y.e')[x:=e] = ? \)
    - The parameter y does not share the same name as x, the variable being substituted for
    - Is \( \lambda y.(e'[x:=e]) \) correct? No…
Variable capture

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 \(y\), as this violates static scoping
- I.e., \((\lambda x.\lambda y.x\ y)\ y \neq \lambda y.y\ y\)

Solution

- \((\lambda x.\lambda y.x\ y)\) is “the same” as \((\lambda x.\lambda z.x\ z)\)
  - Due to alpha conversion
- So alpha-convert \((\lambda x.\lambda y.x\ y)\ y\) to \((\lambda x.\lambda z.x\ z)\ y\) first
  - Now \((\lambda x.\lambda z.x\ z)\ y \rightarrow \lambda z.y\ z\)
Completing the Definition of Substitution

- Recall: we need to define \((\lambda y.e')[x:=e]\)
  - We want to avoid capturing (free) occurrences of \(y\) in \(e\)
  - Solution: alpha-conversion!
    - Change \(y\) to a variable \(w\) that does not appear in \(e'\) or \(e\)
      (Such a \(w\) is called fresh)
    - Replace all occurrences of \(y\) in \(e'\) by \(w\).
    - Then replace all occurrences of \(x\) in \(e'\) by \(e\)!

- Formally:
  \[(\lambda y.e')[x:=e] = \lambda w.((e' [y:=w]) [x:=e]) \text{ (\(w\) is fresh)}\]
Beta-Reduction, Again

Whenever we do a step of beta reduction

• \((\lambda x.e_1)\ e_2 \rightarrow e_1[x:=e_2]\)
• We must alpha-convert variables as necessary
• Usually performed implicitly (w/o showing conversion)

Examples

• \((\lambda x.\lambda y.x\ y)\ y = (\lambda x.\lambda z.x\ z)\ y \rightarrow \lambda z.y\ z \quad // \ y \rightarrow z\)
• \((\lambda x.x\ (\lambda x.x))\ z = (\lambda y.y\ (\lambda x.x))\ z \rightarrow z\ (\lambda x.x) \quad // \ x \rightarrow y\)
OCaml Implementation: Substitution

(* substitute e for y in m *)

let rec subst m y e =
    match m with
    Var x ->
        if y = x then e (* substitute *)
        else m (* don’t subst *)
    | App (e1,e2) ->
        App (subst e1 y e, subst e2 y e)
    | Lam (x,e0) -> ...
let rec subst m y e = match m with ...
  \ Lam (x,e0) ->
  if y = x then m
  else if not (List.mem x (fvs e)) then
    Lam (x, subst e0 y e)
  else
    let z = newvar() in (* fresh *)
    let e0' = subst e0 x (Var z) in
    Lam (z, subst e0' y e)
OCaml Impl: Reduction

let rec reduce e =
    match e with
    | App (Lam (x,e), e2) -> subst e x e2
    | App (e1,e2) ->
      let e1' = reduce e1 in
      if e1' != e1 then App(e1',e2)
      else App (e1,reduce e2)
    | Lam (x,e) -> Lam (x, reduce e)
    | _ -> e

Straight β rule
Reduce lhs of app
Reduce rhs of app
Reduce function body
nothing to do
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 = λx.λy.x
  - false = λx.λy.y
  - if a then b else c
    - Defined to be the λ expression: a b c

Examples
- if true then b else c = (λx.λy.x) b c → (λy.b) c → b
- if false then b else c = (λx.λy.y) b c → (λy.y) c → c
Booleans (cont.)

- Other Boolean operations
  - not = λx.((x false) true)
    - not x = if x then false else true
    - not true → (λx.(x false) true) true → ((true false) true) → false
  - and = λx.λy.((x y) false)
    - and x y = if x then y else false
  - or = λx.λy.((x true) y)
    - or x y = if x then true else y

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

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

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

Encoding of non-negative integers

- $0 = \lambda f.\lambda y.y$
- $1 = \lambda f.\lambda y.f\ y$
- $2 = \lambda f.\lambda y.f\ (f\ y)$
- $3 = \lambda f.\lambda y.f\ (f\ (f\ y))$

i.e., $n = \lambda f.\lambda y.\text{<apply}\ f\ n\ \text{times to}\ y>$

- Formally: $n+1 = \lambda f.\lambda y.f\ (n\ f\ y)$

*(Alonzo Church, of course)*
Operations On Church Numerals

- **Successor**
  - \( \text{succ} = \lambda z. \lambda f. \lambda y. f (z f y) \)
  - \( 0 = \lambda f. \lambda y. y \)
  - \( 1 = \lambda f. \lambda y. f y \)

- **Example**
  - \( \text{succ } 0 = \)
    - \( (\lambda z. \lambda f. \lambda y. f (z f y)) (\lambda f. \lambda y. y) \rightarrow \)
    - \( \lambda f. \lambda y. f ((\lambda f. \lambda y. y) f y) \rightarrow \)
    - \( \lambda f. \lambda y. f ((\lambda y. y) y) \rightarrow \)
    - \( \lambda f. \lambda y. f y \)
    - \( = 1 \)
    
  Since \( (\lambda x. y) z \rightarrow y \)
Operations On Church Numerals (cont.)

- **IsZero?**
  - \(\text{iszero} = \lambda z. z \ (\lambda y. \text{false}) \ \text{true}\)
    
    This is equivalent to \(\lambda z. ((z \ (\lambda y. \text{false})) \ \text{true})\)

- **Example**
  - \(\text{iszero 0} = (\lambda z. z \ (\lambda y. \text{false}) \ \text{true}) \ (\lambda f. \lambda y. y) \rightarrow (\lambda f. \lambda y. y) \ (\lambda y. \text{false}) \ \text{true} \rightarrow (\lambda y. y) \ \text{true} \rightarrow \text{true}\)
  - \(\text{false} \rightarrow (\lambda x. y) \ z \rightarrow y\)
  - \(0 = \lambda f. \lambda y. y\)
Arithmetic Using Church Numerals

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

- Addition
  - \[ M + N = \lambda f.\lambda y.(M f)((N f) y) \]
  
  Equivalently:
  \[ + = \lambda M.\lambda N.\lambda f.\lambda y.(M f)((N f) y) \]
  
  In prefix notation (+ M N)

- Multiplication
  - \[ M \times N = \lambda f.(M (N f)) \]
  
  Equivalently:
  \[ \times = \lambda M.\lambda N.\lambda f.\lambda y.(M (N f)) y \]
  
  In prefix notation (* M N)
Arithmetic (cont.)

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

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

- $1 = \lambda f.\lambda y.f \, y$
- $2 = \lambda f.\lambda y.f \, (f \, y)$
Looping & Recursion

- 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 Fixpoint Combinator

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

Then

\[ Y \ F = \]

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

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

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

\[ = F (Y \ F) \]

\[ Y \ F \text{ is a } \textit{fixed point} \text{ (aka “fixpoint”) of } F \]

Thus \[ Y \ F = F (Y \ F) = F (F (Y \ F)) = ... \]

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

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

- 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

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

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

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

- $t ::= \text{int} \mid t \to t$
  - int is the type of integers
  - $t_1 \to t_2$ is the type of a function
    - That takes arguments of type $t_1$ and returns result of type $t_2$
  - $t_1$ is the domain and $t_2$ is the range
  - Notice this is a recursive definition
    - So we can give types to higher-order functions
Types are limiting

- STLC will reject some terms as ill-typed, even if they will not produce a run-time error
  - Cannot type check Y in STLC
    - Or in Ocaml, for that matter!
- Surprising theorem: All (well typed) simply-typed lambda calculus terms are strongly normalizing
  - They will terminate
  - Proof is not by straightforward induction
    - Applications “increase” term size
Summary

- Lambda calculus shows issues with
  - Scoping
  - Higher-order functions
  - Types

- Useful for understanding how languages work