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

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

Lambda Calculus (λ-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	variable
| λx.e	function
| e e	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)

- λx.e is like `(fun x -> e)` in OCaml
- That's it! Nothing but higher-order functions

Three Conveniences

- "Syntactic sugar" for local declarations
  - let x = e1 in e2 is short for (λx.e2) e1
- Scope of λ extends as far right as possible
  - Subject to scope delimited by parentheses
  - λx. λy.x y is same as λx.(λy.(x y))
- Function application is left-associative
  - x y z is (x y) z
  - Same rule as OCaml

OCaml implementation

- e ::= x
  | λx.e
  | e e

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

y	`Var "y"`
λx.x	`Lam ("x", Var "x")`
λx.λy.x y	`Lam ("x", (Lam ("y", App (Var "x", Var "y")`
(λx.λy.x y) λx.x x	`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 (λx.e1) e2
  - Evaluate e1 with x replaced by e2
- This application is called beta-reduction
  - (λx.e1) e2 → e1[x:=e2]
    - e1[x:=e2] is e1 with occurrences of x replaced by e2
    - 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

- (λx.λz.x z) y

  → (λx.(λz.(x z))) y // since λ extends to right

  → (λx.(λz.(x z))) y // apply (λx.e1) e2 → e1[x:=e2]

  // where e1 = λz.(x z), e2 = y

  → λz.(y z) // final result

Parameters
- Formal
- Actual

- Equivalent OCaml code
  - (fun x -> (fun z -> (x z))) y → fun z -> (y z)

Lambda Calculus Examples

- (λx.x) z → z
- (λx.y) z → y
- (λx.x y) z → z y
  - A function that applies its argument to y

Lambda Calculus Examples (cont.)

- (λx.x y) (λz.z) → (λz.z) y → y
- (λx.λy.x y) z → λy.z y
  - A curried function of two arguments
  - Applies its first argument to its second
- (λx.λy.x y) (λz.zz) x →(λy.(λz.zz)y)x → (λz.zz)x → xx

Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - 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 (λx.x (λy.y))
  - Renaming bound variables consistently is allowed
    - This is called alpha-renaming or alpha conversion
  - Ex. λx.x = λy.y = λz.z   λy.λx.y = λz.λx.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
  - (λx.e')[x:=e] = λx.e'
    - In λx.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'
  - (λy.e')[x:=e] = ?
    - The parameter y does not share the same name as x, the variable being substituted for
    - Is λy.(e'[x:=e]) correct? No…

Variable capture

- How about the following?
  - (λx.λy.x y) y → ?
  - 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., (λx.λy.x y) y ≠ λy.y y
- Solution
  - (λx.λy.x y) is "the same" as (λx.λz.x z)
    - Due to alpha conversion
  - So alpha-convert (λx.λy.x y) y to (λx.λz.x z) y first
    - Now (λx.λz.x z) y → λz.y z

Completing the Definition of Substitution

- Recall: we need to define (λy.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:
  (λy.e')[x:=e] = λw.((e' [y:=w]) [x:=e]) (w is fresh)

Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - (λx.e1) e2 → e1[x:=e2]
  - We must alpha-convert variables as necessary
  - Usually performed implicitly (w/o showing conversion)
- Examples
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z // y → z
  - (λx.x (λx.x)) z = (λy.y (λx.x)) z → z (λx.x) // x → 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) -> …
```


OCaml Impl: Substitution (cont'd)

```
(* substitute e for y in m *)
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)
```

Shadowing blocks substitution

Safe: no capture possible

Might capture; need to α-convert

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) = 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>
  - Formally: n+1 = λf.λy.f (n f y)

*(Alonzo Church, of course)

Operations On Church Numerals

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

    (λz.λf.λy.f (z f y)) (λf.λy.y) →

    λf.λy.f ((λf.λy.y) f y) →

    λf.λy.f ((λy.y) y) →

    λf.λy.f y

    = 1

- 0 = λf.λy.y
- 1 = λf.λy.f y

Since (λx.y) z → y

Operations On Church Numerals (cont.)

- IsZero?
  - iszero = λz.z (λy.false) true

    This is equivalent to λz.((z (λy.false)) true)
- Example
  - iszero 0 =

    (λz.z (λy.false) true) (λf.λy.y) →

    (λf.λy.y) (λy.false) true →

    (λy.y) true →

    true

- 0 = λf.λy.y

Since (λx.y) z → y

Arithmetic Using Church Numerals

- If M and N are numbers (as λ expressions)
  - Can also encode various arithmetic operations
- Addition
  - M + N = λf.λy.(M f)((N f) y)

    Equivalently: + = λM.λN.λf.λy.(M f)((N f) y)
    - In prefix notation (+ M N)
- Multiplication
  - M * N = λf.(M (N f))

    Equivalently: * = λM.λN.λf.λy.(M (N f)) y
    - In prefix notation (* M N)

Arithmetic (cont.)

- Prove 1+1 = 2
  - 1+1 = λx.λy.(1 x)((1 x) y) =
  - λx.λy.((λf.λy.f y) x)((1 x) y) →
  - λx.λy.(λy.x y)((1 x) y) →
  - λx.λy.x ((1 x) y) →
  - λx.λy.x (((λf.λy.f y) x) y) →
  - λx.λy.x ((λy.x y) y) →
  - λx.λy.x (x y) = 2

- 1 = λf.λy.f y
- 2 = λf.λy.f (f y)

- With these definitions
  - Can build a theory of arithmetic

Looping & Recursion

- 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 Fixpoint Combinator

Y = λf.(λx.f (x x)) (λx.f (x x))

- Then

  Y F =
  (λf.(λx.f (x x)) (λx.f (x x))) F →
  (λx.F (x x)) (λx.F (x x)) →
  F ((λx.F (x x)) (λx.F (x x)))
  = F (**Y** F)

- **Y** F is a *fixed point* (aka "fixpoint") of F
- Thus **Y** F = F (**Y** F) = F (F (**Y** F)) = ...
  - We can use **Y** to achieve recursion for F

Example

fact = λf. λn.if n = 0 then 1 else n * (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** fact) 1 = (fact (**Y** fact)) 1
→ if 1 = 0 then 1 else 1 * ((**Y** fact) 0)
→ 1 * ((Y fact) 0)
→ 1 * (fact (Y fact) 0)
→ 1 * (if 0 = 0 then 1 else 0 * ((**Y** fact) (-1))
→ 1 * 1 → 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 | x | λx:t.e | 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
- 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 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

Type Checking STLC

- Can define a type checking algorithm using inference rules
  - In the same style as the operational semantics we saw last time. Defines a *judgment*

$$\Gamma \vdash e : t$$

which is read *e has type t in environment Γ where*

- Γ ::= • | Γ, x:t
  - Γ is a type environment, mapping ids to types

Type Environments

- • is the empty type environment
  - undefined for all ids
- x:t is the environment that says x has type t
  - and is undefined for all other ids
- If Γ and Γ' are environments then Γ, Γ' is the environment defined as follows

$$(\Gamma, \Gamma')(id) = \begin{cases} \Gamma'(id) & \text{if } \Gamma'(id) \text{ defined} \\ \Gamma\,(id) & \text{if } \Gamma'(id) \text{ undefined but } \Gamma(id) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

  - Idea: Γ' "overridess" definitions in Γ
  - For brevity, can write •, Γ as just Γ

Type Checking STLC Inference Rules

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \frac{\Gamma, x{:}t \vdash e : t'}{\Gamma \vdash \lambda x{:}t.e : t \rightarrow t'}$$

$$\frac{-}{\Gamma \vdash n : int} \qquad \frac{\Gamma \vdash e1 : t \rightarrow t' \quad \Gamma \vdash e2 : t}{\Gamma \vdash e1\ e2 : t'}$$

These rules work the same way as the operational semantics rules: "Proofs" are trees that string them together to prove that an expression has a particular type

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
    - Function applications "increase" term size

Summary

- Lambda calculus shows issues with
  - Scoping
  - Higher-order functions
  - Types
- Useful for understanding how languages work