CMSC 330: Organization of Programming Languages

Lambda Calculus

Quiz #7

Beta reducing the following term produces what result?

$$\lambda x.(\lambda y.\ y\ y)\ w\ z$$

a) λx. w w z

b) λx. w z

c) w z

d) Does not reduce

Quiz #7

Beta reducing the following term produces what result?

λx.(λy. y y) w z

a) λx. w w z

b) λx. w z

c) w z

d) Does not reduce

Lambda Calc, Impl in OCaml

- 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")))
```

Quiz #8

What is this term's AST?

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

λx.x x

A. `App (Lam ("x", Var "x"), Var "x")`
B. `Lam (Var "x", Var "x", Var "x")`
C. `Lam ("x", App (Var "x",Var "x"))`
D. `App (Lam ("x", App ("x", "x")))`

Quiz #8

What is this term's AST?

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

λx.x x

A. `App (Lam ("x", Var "x"), Var "x")`
B. `Lam (Var "x", Var "x", Var "x")`
C. `Lam ("x", App (Var "x",Var "x"))`
D. `App (Lam ("x", App ("x", "x")))`

OCaml Implementation: Substitution

```
(* substitute e for y in m--   m[y:=e]      *)
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--   m[y:=e]   *)
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

CBV, L-to-R Reduction with Partial Eval

```
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

Another Way to Avoid Capture

- Another way to avoid accidental variable capture is to use the “Barendregt Convention”: gives everything ‘fresh’ names.
  - If every name is unique, no chance of variable capture
  - Simple, but not great for performance as you have to do it after every beta-reduction!

Quick Recap on LC

- Despite its simplicity (3 AST nodes and a handful of small-step rules), LC is Turing Complete
- Any function that can be evaluated on a Turing machine can be encoded into LC (and vice-versa)
  - But we'll have to come up with the encodings!
- To *prove* that it is Turing Complete we have to map every possible Turing Machine to LC
  - We won't be doing that

The Power of Lambdas

- To give a sense of how one can encode various constructs into LC we'll be looking at some concrete examples:
  - Let bindings
  - Booleans
  - Pairs
  - Natural numbers & arithmetic
  - Looping

Let bindings

- Local variable declarations are like defining a function and applying it immediately (once):
  - let x = e1 in e2 = (λx.e2) e1

- Example
  - let x = (λy.y) in x x = (λx.x x) (λy.y)

  where

  (λx.x x) (λy.y) → (λx.x x) (λy.y) → (λy.y) (λy.y) → (λy.y)

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 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* = *x* false true = 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

Quiz #9

What is the lambda calculus encoding of xor x y?

- xor true true = xor false false = false
- xor true false = xor false true = true

- x x y
- x (y true false) y
- x (y false true) y
- y x y

true = λx.λy.x
false = λx.λy.y
if a then b else c = a b c
not = λx.x false true

Quiz #9

What is the lambda calculus encoding of xor x y?

- xor true true = xor false false = false
- xor true false = xor false true = true

- x x y
- x (y true false) y
- **x (y false true) y**
- y x y

true = λx.λy.x
false = λx.λy.y
if a then b else c = a b c
not = λx.x false true

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)

Quiz #10

n = λf.λy.<*apply* f *n times to* y>

What OCaml type could you give to a Church-encoded numeral?

- (’a -> ‘b) -> ‘a -> ‘b
- (‘a -> ‘a) -> ‘a -> ‘a
- (‘a -> ‘a) -> ‘b -> int
- (int -> int) -> int -> int

Quiz #10

n = λf.λy.<*apply* f *n times to* y>

What OCaml type could you give to a Church-encoded numeral?

- (’a -> ‘b) -> ‘a -> ‘b
- **(‘a -> ‘a) -> ‘a -> ‘a**
- (‘a -> ‘a) -> ‘b -> int
- (int -> int) -> int -> int

Operations On Church Numerals

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

- 0 = λf.λy.y
- 1 = λf.λy.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

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

Arithmetic Using Church Numerals

- What about subtraction?
  - Easy once you have ‘predecessor’, but...
  - Predecessor is very difficult!
- Story time:
  - One of Church’s students, Kleene (of Kleene-star fame) was struggling to think of how to encode ‘predecessor’, until it came to him during a trip to the dentists office.
  - Take from this what you will
- Wikipedia has a great derivation of ‘predecessor’, not enough time today.

Looping+Recursion

- So far we have avoided self-reference, so how does recursion work?
- We can construct a lambda term that 'replicates' itself:
  - Define D = λx.x x, then
    - D D = (λx.x x) (λx.x x) → (λx.x x) (λx.x x) = D D
  - D D is an infinite loop
- We want to generalize this, so that we can make use of 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

Factorial 4=?

```
(Y G) 4
 G (Y G) 4
(λr.λn.(if n = 0 then 1  else n × (r (n-1)))) (Y G) 4
(λn.(if n = 0 then 1 else n × ((Y G) (n-1)))) 4
if 4 = 0 then 1 else 4 × ((Y G) (4-1))
4 × (G (Y G) (4-1))
4 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (4-1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3-1)))
4 × (3 × (G (Y G) (3-1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (3-1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2-1))))
4 × (3 × (2 × (G (Y G) (2-1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (2-1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1-1)))))
4 × (3 × (2 × (1 × (G (Y G) (1-1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (1-1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0-1))))))
4 × (3 × (2 × (1 × (1))))
24
```

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 t 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

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, at least not as written earlier.
- Surprising theorem: All (well typed) simply-typed lambda calculus terms are strongly normalizing
  - A normal form is one that cannot be reduced further
    - A value is a kind of normal form
  - Strong normalization means STLC terms always terminate
    - Proof is *not* by straightforward induction: Applications "increase" term size

Summary

- Lambda calculus is a core model of computation
  - We can encode familiar language constructs using only functions
    - These encodings are enlightening – make you a better (functional) programmer
- Useful for understanding how languages work
  - Ideas of types, evaluation order, termination, proof systems, etc. can be developed in lambda calculus,
    - then scaled to full languages