CMSC 330: Organization of Programming Languages

Lambda Calculus Encodings

The Power of Lambdas

- Despite its simplicity, the lambda calculus is quite expressive: it 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 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 #1

What is the lambda calculus encoding of xor x y?

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

A. x x y
B. x (y true false) y
C. x (y false true) y
D. 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 #1

What is the lambda calculus encoding of xor x y?

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

A. x x y
B. x (y true false) y
C. **x (y false true) y**
D. 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 #2

n = λf.λy.<*apply* f *n times to* y>

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

A. (’a -> ‘b) -> ‘a -> ‘b
B. (‘a -> ‘a) -> ‘a -> ‘a
C. (‘a -> ‘a) -> ‘b -> int
D. (int -> int) -> int -> int

Quiz #2

n = λf.λy.<*apply* f *n times to* y>

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

A. (’a -> ‘b) -> ‘a -> ‘b
B. **(‘a -> ‘a) -> ‘a -> ‘a**
C. (‘a -> ‘a) -> ‘b -> int
D. (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 → Since (λx.y) z → y

    true

  - 0 = λf.λy.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
  - $M * N = \lambda f.M\ N\ f$

    Equivalently: $* = \lambda M.\lambda N.\lambda f.\lambda y.M\ N\ f\ y$
    - In prefix notation (* M N)

Arithmetic (cont.)

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

Call-by-name vs. Call-by-value

- Sometimes we have a choice about where to apply beta reduction. Before call (i.e., argument):
  - (λz.z) ((λy.y) x) → (λz.z) x → x
- Or after the call:
  - (λz.z) ((λy.y) x) → (λy.y) x → x
- The former strategy is called call-by-value
  - Evaluate any arguments before calling the function
- The latter is called call-by-name
  - Delay evaluating arguments as long as possible

Confluence

- No matter what evaluation order you choose, you get the same answer
  - Assuming the evaluation always terminates
  - Surprising result!
- However, termination behavior differs between call-by-value and call-by-name
  - if true then true else (D D) → true under call-by-name
    - true true (D D) = (λx.λy.x) true (D D) → (λy.true) (D D) → true
  - if true then true else (D D) → … under call-by-value
    - (λx.λy.x) true (D D) → (λy.true) (D D) → (λy.true) (D D) → … never terminates

Quiz #3

Y is a fixed point combinator under which evaluation order?

A. Call-by-value
B. Call-by-name
C. Both
D. Neither

Y = λf.(λx.f (x x)) (λx.f (x x))

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)

Quiz #3

Y is a fixed point combinator under which evaluation order?

A. Call-by-value
B. **Call-by-name**
C. Both
D. Neither

Y = λf.(λx.f (x x)) (λx.f (x x))

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)

In CBV, we expand
Y F = F (Y F) = F (F (Y F)) … indefinitely, for any F

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!
- 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