CS 314: Principles 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*
- Examples
  - Let bindings
  - Booleans
  - Pairs
  - Natural numbers & arithmetic
  - Looping

*To show Turing completeness we have to map every Turing machine to lambda calculus term. We are not doing that here. Rather, we are showing how typical PL constructs can be represented in lambda calculus, to show what it can do

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) → (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) →
      if true then false else true → false
  - and = λx.λy.x y false
    - and x y = x y false = if x then y else false
  - or = λx.λy.x true y
    - or x y = x true 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.f (f (f (f .. (f 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)
- 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

Operations On Church Numerals (cont.)

- IsZero?
  - iszero = 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

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)

```
M + N = λf.λy.M f (N f y)
      = λx.λy.M f (N x y)
```

Arithmetic (cont.)

- Prove 1+1 = 2
  - 1+1 = λx.λy.(1 x) (1 x y) =
  - λx.λy.((λf.λy.f y) x) (1 f 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
- With these definitions
  - Can build a theory of arithmetic

- 1 = λf.λy.f y
- 2 = λf.λy.f (f y)

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 (CBV)
  - Evaluate any arguments before calling the function
- The latter is called call-by-name (CBN)
  - Delay evaluating arguments as long as possible

Confluence

- No matter what evaluation order (or combination) you choose, you get the same answer
  - Assuming the evaluation always terminates

Partial Evaluation

- It is also possible to evaluate within a function (without calling it):
  - (λy.(λz.z) y x) → (λy.y x)
- Called partial evaluation
  - Can combine with CBN or CBV
  - In practical languages, this evaluation strategy is employed in a limited way, as compiler optimization

```
int foo(int x) {
  return 0+x;
}
```

→

```
int foo(int x) {
  return x;
}
```

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

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