CMSC 330: Organization of Programming Languages

Lambda Calculus and Types

Introduction

- We've seen that several language conveniences aren't strictly necessary
  - Multi-argument functions: use currying or tuples
  - Loops: use recursion
  - Side-effects: we don't need them either
- Goal: come up with a "core" language that's as small as possible and still Turing-complete
  - This will give a way of illustrating important language features and algorithms
- One solution (there are others) is lambda calculus

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

Lambda Calculus (λ-calculus)

- Proposed in 1930s by Alonzo Church and Stephen Cole Kleene
- A formal system designed to investigate functions and recursion, and for exploration of foundations of mathematics
- Now used as a tool for investigating computability.
  - It's also the basis of functional programming languages such as Lisp, Scheme, ML, OCaml, Haskell…

Lambda Calculus

- A lambda calculus expression is defined as

e → x
  | λx.e
  | e e

- λx.e is like `(fun x -> e)` in OCaml
- That's it! Higher-order functions is all there is

Three Conveniences

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

Semantics of Function Application

- All we've got are functions, so all we can do is call them
- To evaluate (λx.e1) e2, evaluate e1 with x bound to e2
- This application is called *beta-reduction*, and the rule describing it is:

  (λx.e1) e2 → e1[x/e2]
  - e1[x/e2] means e1 where occurrences of x are replaced by e2
    - This is slightly different than the environments we saw for Ocaml- do substitutions to replace formals with actuals, instead of using the environment to map formals to actuals
  - Reductions are allowed to occur anywhere in a term

Beta reduction examples

- (λx.x) z →
- (λx.y) z →
- (λx.x y) z →
- (λx.x y) (λz.z) →
- (λx.λy.x y) z →
  - Equivalent OCaml code:
    (fun x -> (fun y -> (x y))) z → fun y -> (z y)
- (λx.λy.x y) (λz.z z) x →

Static Scoping

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - The rightmost "x" refers to the second binding (the inner "x" hides or shadows the outer "x").
  - 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

Static Scoping (cont'd)

- 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
  - I.e., (λx.λy.x y) y ≠ λy.y y
- This function is "the same" as (λx.λz.x z)

Beta-Reduction, Again

- Whenever we do a step of beta reduction (using the rule (λx.e1) e2 → e1[x/e2]), alpha-convert variables as necessary
- Examples:
  - (λx.x (λx.x)) z = (λx.x (λy.y)) z → z (λy.y)
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z

Encodings

- It turns out that this language is Turing complete
- That means we can encode any computation we want in it, if we're sufficiently clever.

Encoding Booleans

- Booleans and conditionals can be encoded or represented, using only the lambda calculus, as follows:

  true = λx.λy.x

  false = λx.λy.y

  if a then b else c = a b c

- Examples:
  - if true then b else c →
  - if false then b else c →

Booleans (cont.)

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

- Given these operations we can build up a logical inference system

Encoding Pairs

(a,b) = λx.if x then a else b

fst = λx.x true (returns first component)

snd = λx.x false (returns second component)

Encoding Natural Numbers (Church*)

*(Named after Alonzo Church)

0 = λx.λy.y

1 = λx.λy.x y

2 = λx.λy.x (x y)

i.e., n = λx.λy.<apply x to y n times >

succ = λz.λx.λy.x (z x y)

iszero = λz.z (λy.false) true

- Recall that this is equivalent to λz.((z (λy.false)) true)

Arithmetic Using Church Numerals

- If M and N are numbers (as λ expressions) we can also encode various arithmetic operations
- Addition
  - M + N = λx.λy.(M x)((N x) y)

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

    Equivalently: * = λM.λN.λx.(M (N x))
    - In prefix notation (* M N)

Arithmetic (cont.)

1 = λf.λy.f y

2 = λf.λy.f (f y)

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

many implicit alpha conversions

- With these definitions we can build a theory of arithmetic

Repetition

- 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 repetition

The "Paradoxical" 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)

- Thus Y F = F (Y F) = F (F (Y 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

- Using encodings we can represent pretty much anything we have in a "real" language
  - But programs would be pretty slow if we really implemented things this way
  - In practice, we use richer languages that include built-in primitives
- Lambda calculus shows all the issues with scoping and higher-order functions
- It's useful for understanding how languages work