CMSC 330: Organization of Programming Languages

Lambda Calculus and Types

Lambda Calculus

- A lambda calculus expression is defined as

e ::= x	variable
| λx.e	function
| e e	function application

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

Beta-Reduction, Again

- Whenever we do a step of beta reduction...
  - (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...

Booleans

The lambda calculus was created by logician Alonzo Church in the 1930's to formulate a mathematical logical system

true = $\lambda x.\lambda y.x$

false = $\lambda x.\lambda y.y$

if a then b else c is defined to be the λ expression: a b c

- Examples:
  - if true then b else c $\rightarrow$ $(\lambda x.\lambda y.x)$ b c $\rightarrow$ $(\lambda y.b)$ c $\rightarrow$ b
  - if false then b else c $\rightarrow$ $(\lambda x.\lambda y.y)$ b c $\rightarrow$ $(\lambda y.y)$ c $\rightarrow$ c

Booleans (continued)

Other Boolean operations:

- not = $\lambda x.((x \text{ false}) \text{ true})$
- not true $\rightarrow$ $\lambda x.((x \text{ false}) \text{ true})$ true $\rightarrow$
  ((true false) true) $\rightarrow$ false
- and = $\lambda x.\lambda y.((xy) \text{ false})$
- or = $\lambda x.\lambda y.((x \text{ true}) \; y)$
- Show not, and and or have the desired properties, …
- Given these operations, can build up a logical inference system

Pairs

(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*)

*(Named after Alonzo Church, developer of lambda calculus)

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>

succ = λz.λf.λy.f (z f y)
iszero = λg.g (λy.false) true
 – Recall that this is equivalent to λg.((g (λy.false)) true)

Natural Numbers (cont'd)

- Examples:

  succ 0 =
  (λz.λf.λy.f (z f y)) (λf.λy.y) →
  λf.λy.f ((λf.λy.y) f y) →
  λf.λy.f y = 1

  iszero 0 =
  (λz.z (λy.false) true) (λf.λy.y) →
  (λf.λy.y) (λy.false) true →
  (λy.y) true →
  true

Arithmetic defined

- Addition, if M and N are integers (as λ expressions):
  M + N = λx.λy.(M x)((N x) y)
  Equivalently: + = λM.λN.λx.λy.(M x)((N x) y)
- Multiplication: M * N = λx.(M (N x))
- 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
- With these definitions, can build a theory of integer arithmetic.

Looping

- 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 "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

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 ...
    - 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

What is a Type System?

- A *type system* is some mechanism for distinguishing good programs from bad
  - Good = well typed
  - Bad = ill typed or not typable; has a *type error*
- Examples
  - 0 + 1 // well typed
  - false 0 // ill-typed; can't apply a boolean

Static versus Dynamic Typing

- In a *static type system*, we guarantee at compile time that all program executions will be free of type errors
  - OCaml and C have static type systems
- In a *dynamic type system*, we wait until runtime, and halt a program (or raise an exception) if we detect a type error
  - Ruby has a dynamic type system
- Java, C++ have a combination of the two

Simply-Typed Lambda Calculus

- e ::= n | x | λx:t.e | e e
  - We've added integers n as primitives
    - Without at least two disinct types (integer and function), can't have any 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 a result of type t2
  - t1 is the *domain* and t2 is the *range*
  - Notice this is a recursive definition, so that we can give types to higher-order functions

Type Judgments

- We will construct a type system that proves *judgments* of the form

$$A \vdash e : t$$

  - "In type environment A, expression e has type t"
- If for a program e we can prove that it has some type, then the program type checks
  - Otherwise the program has a type error, and we'll reject the program as bad

Type Environments

- A *type environment* is a map from variables names to their types
  - Just like in our operational semantics for Scheme
- • is the empty type environment
- A, x:t is just like A, except x now has type t
- When we see a variable in the program, we'll look up its type in the environment

Type Rules

$e ::= n \mid x \mid \lambda x{:}t.e \mid e\ e$

$$\frac{}{A \vdash n : \text{int}} \qquad \frac{x \in A}{A \vdash x : A(x)}$$

$$\frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\ e' : t'}$$

Example

$A = + : \text{int} \to \text{int} \to \text{int}$

$B = A, x : \text{int}$

$$\dfrac{\dfrac{\dfrac{\dfrac{B \vdash + : i \to i \to i \qquad B \vdash x : \text{int}}{B \vdash +\ x : \text{int} \to \text{int}} \qquad B \vdash 3 : \text{int}}{B \vdash +\ x\ 3 : \text{int}}}{A \vdash (\lambda x{:}\text{int}.+\ x\ 3) : \text{int} \to \text{int}} \qquad A \vdash 4 : \text{int}}{A \vdash (\lambda x{:}\text{int}.+\ x\ 3)\ 4 : \text{int}}$$

Discussion

- The type rules are a kind of logic for reasoning about types of programs
  - The tree of judgments we just saw is a kind of *proof* in this logic that the program has a valid type
- So the *type checking* problem is like solving a jigsaw puzzle
  - Can we apply the rules to a program in such a way as to produce a typing proof?
  - It turns out we can easily decide whether or not we can do this.

An Algorithm for Type Checking

(Write this in OCaml!)

TypeCheck : type env × expression → type

```
TypeCheck(A, n) = int
TypeCheck(A, x) = if x in A then A(x) else fail
TypeCheck(A, λx:t.e) =
  let t' = TypeCheck((A, x:t), e) in t → t'
TypeCheck(A, e1 e2) =
  let t1 = TypeCheck(A, e1) in
  let t2 = TypeCheck(A, e2) in
   if dom(t1) = t2 then range(t1) else fail
```

Type Inference

- We could extend the rules to show how a language could figure out, even if types aren't specified, what the types of everything are in a program
  - Can you believe there are languages which can actually do this?
- We could do these things, but we actually won't.

Summary

- Lambda calculus shows all the issues with scoping and higher-order functions
- It's useful for understanding how languages work

Practice

- Reduce the following:
  - (λx.λy.x y y) (λa.a) b
  - (or true) (and true false)
  - (* 1 2)   (* m n = λM.λN.λx.(M (N x)))

- Derive and prove the type of:
  - (λf:int->int.λn:int.f n) (λx:int. 3 + x) 6
  - λx:int->int->int. λy:int->int. λz:int.x z (y z)