

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)

```
let fst (x,y) = x
```

```
let fst x y = x
```

```
let fst p =  
  match p with  
  (x,y) -> x
```

```
let fst x =  
  fun y -> x
```

Introduction

- Loops (use recursion)
- Side-effects

```
r = 0;  
for (i = 0; i < n; i++) {  
  r += i;  
}  
  
let rec sum i r =  
  if i >= n then r  
  else sum (i+1) (r+i)  
in  
sum 0 0
```

Introduction

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

Lambda Calculus Syntax

- A lambda calculus expression is defined as

$e ::= x$	variable
$\lambda x.e$	function
$e e$	function application

- $\lambda x.e$ is like `(fun x -> e)` in OCaml
- That's it! All there is is higher-order functions

Conventions

- The scope of λ extends as far to the right as possible

$\lambda x. \lambda y. x y$ is $\lambda x. (\lambda y. (x y))$

- Function application is left-associative

$x y z$ is $(x y) z$

- Same rule as OCaml

Operational Semantics

An interpreter in "math"

- All we've got are functions, so all we can do is call them
- To evaluate $(\lambda x.e1) e2$
 - Evaluate $e1$ with x bound to $e2$

beta-reduction

$$(\lambda x.e1) e2 \rightarrow e1[x/e2]$$

- $e1[x/e2]$ is $e1$ where occurrences of x are replaced by $e2$
- Slightly different than the environments we saw for OCaml
 - apply a substitution instead of carry an environment
- We allow reductions to occur anywhere in a term

Examples

- $(\lambda x.x) z \rightarrow z$
- $(\lambda x.y) z \rightarrow y$
- $(\lambda x.x y) z \rightarrow zy$
 - A function that applies its argument to y
- $(\lambda x.x y) (\lambda z.z) \rightarrow (\lambda z.z) y \rightarrow y$
- $(\lambda x.\lambda y.x y) z \rightarrow \lambda y.z y$
 - A curried function of two arguments that applies its first argument to its second
- $(\lambda x.\lambda y.x y) (\lambda z.zz) x \rightarrow \lambda y.((\lambda z.zz)y)x \rightarrow (\lambda z.zz)x \rightarrow xx$

Syntactic sugar for local declarations

let $x = e1$ in $e2$

can be written as

$(\lambda x. e2) e1$

Static Scoping and Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
 - $(\lambda x. x (\lambda x. x)) z \rightarrow ?$
 - The rightmost “ x ” refers to the second binding
 - This is a function that takes its argument and applies it to the identity function
- This function is “the same” as $(\lambda x. x (\lambda y. y))$
 - Renaming bound variables consistently is allowed

alpha-conversion (alpha-renaming)

Ex. $\lambda x. x = \lambda y. y = \lambda z. z \quad \lambda y. \lambda x. y = \lambda z. \lambda x. z$

Static Scoping (cont'd)

- How about the following?
 - $(\lambda x. \lambda y. x y) y \rightarrow ?$
 - When we replace y inside, we don't want it to be “captured” by the inner binding of y
- This function is “the same” as $(\lambda x. \lambda z. x z)$

Beta-Reduction, Again

- Whenever we do a step of beta reduction...
 - $(\lambda x. e1) e2 \rightarrow e1[x/e2]$
 - ...alpha-convert variables as necessary
- Examples:
 - $(\lambda x. x (\lambda x. x)) z = (\lambda x. x (\lambda y. y)) z \rightarrow z (\lambda y. y)$
 - $(\lambda x. \lambda y. x y) y = (\lambda x. \lambda z. x z) y \rightarrow \lambda z. y z$

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

$\text{true} = \lambda x.\lambda y.x$

$\text{false} = \lambda x.\lambda y.y$

$\text{if } a \text{ then } b \text{ else } c = a \ b \ c$ (the λ expression)

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

Booleans (continued)

Other Boolean operations:

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

Pairs

$(a,b) = \lambda x.\text{if } x \text{ then } a \ \text{else } b$

$\text{fst} = \lambda f.f \ \text{true}$

$\text{snd} = \lambda f.f \ \text{false}$

- Examples:
 - $\text{fst } (a,b) = (\lambda f.f \ \text{true}) \ (\lambda x.\text{if } x \ \text{then } a \ \text{else } b) \rightarrow (\lambda x.\text{if } x \ \text{then } a \ \text{else } b) \ \text{true} \rightarrow \text{if true then } a \ \text{else } b \rightarrow a$
 - $\text{snd } (a,b) = (\lambda f.f \ \text{false}) \ (\lambda x.\text{if } x \ \text{then } a \ \text{else } b) \rightarrow (\lambda x.\text{if } x \ \text{then } a \ \text{else } b) \ \text{false} \rightarrow \text{if false then } a \ \text{else } b \rightarrow b$

Natural Numbers (Church*)

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

$$0 = \lambda f.\lambda y.y$$

$$1 = \lambda f.\lambda y.f y$$

$$2 = \lambda f.\lambda y.f (f y)$$

$$3 = \lambda f.\lambda y.f (f (f y))$$

i.e., $n = \lambda f.\lambda y.<\text{apply } f \text{ } n \text{ times to } y>$

$$\text{succ} = \lambda z.\lambda f.\lambda y.f (z f y)$$

$$\text{iszero} = \lambda g.g (\lambda y.\text{false}) \text{ true}$$

– Recall that this is equivalent to $\lambda g.((g (\lambda y.\text{false})) \text{ true})$

Natural Numbers (cont'd)

- Examples:

$$\text{succ } 0 =$$

$$(\lambda z.\lambda f.\lambda y.f (z f y)) (\lambda f.\lambda y.y) \rightarrow$$

$$\lambda f.\lambda y.f ((\lambda f.\lambda y.y) f y) \rightarrow$$

$$\lambda f.\lambda y.f y = 1$$

$$\text{iszero } 0 =$$

$$(\lambda z.z (\lambda y.\text{false}) \text{ true}) (\lambda f.\lambda y.y) \rightarrow$$

$$(\lambda f.\lambda y.y) (\lambda y.\text{false}) \text{ true} \rightarrow$$

$$(\lambda y.y) \text{ true} \rightarrow$$

$$\text{true}$$

Arithmetic defined

- Addition, if M and N are integers (as λ expressions):

$$M + N = \lambda x.\lambda y.(M x)((N x) y)$$

$$\text{Equivalently: } + = \lambda M.\lambda N.\lambda x.\lambda y.(M x)((N x) y)$$

- Multiplication: $M * N = \lambda x.(M (N x))$

- Prove $1+1 = 2$.

$$1+1 = \lambda x.\lambda y.(1 x)((1 x) y) \rightarrow$$

$$\lambda x.\lambda y.((\lambda x.\lambda y.x y) x)((\lambda x.\lambda y.x y) x) y \rightarrow$$

$$\lambda x.\lambda y.(\lambda y.x y)((\lambda x.\lambda y.x y) x) y \rightarrow$$

$$\lambda x.\lambda y.(\lambda y.x y)((\lambda y.x y) y) \rightarrow$$

$$\lambda x.\lambda y.x ((\lambda y.x y) y) \rightarrow$$

$$\lambda x.\lambda y.x (x y) = 2$$

- With these definitions, can build a theory of integer arithmetic.

What else?

- What about looping or recursion?

let rec fact n =

 if n = 0 then 1

 else n * fact (n-1)

Looping

- Define $D = \lambda x.x x$
- Then
 - $D D = (\lambda x.x x) (\lambda x.x x) \rightarrow (\lambda x.x x) (\lambda x.x x) = D D$
- So $D D$ is an infinite loop
 - In general, *self application* is how we get looping

The “Paradoxical” Combinator

$$Y = \lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))$$

- Then
 - $Y F =$
 - $(\lambda f.(\lambda x.f (x x)) (\lambda x.f (x x))) F \rightarrow$
 - $(\lambda x.F (x x)) (\lambda x.F (x x)) \rightarrow$
 - $F ((\lambda x.F (x x)) (\lambda x.F (x x)))$
 - $= F (Y F)$
- Thus $Y F = F (Y F) = F (F (Y F)) = \dots$

Example

$$\text{fact} = \lambda f. \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (f (n-1))$$

- 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

$$(Y \text{ fact}) 1 = (\text{fact } (Y \text{ fact})) 1$$

- if $1 = 0$ then 1 else $1 * ((Y \text{ fact}) 0)$
- $1 * ((Y \text{ fact}) 0)$
- $1 * (\text{fact } (Y \text{ fact}) 0)$
- $1 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * ((Y \text{ fact}) (-1)))$
- $1 * 1 \rightarrow 1$

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
 - $\text{false} = \lambda x.\lambda y.y$
 - $0 = \lambda x.\lambda y.y$
- Since everything is encoded as a function...
 - We can easily misuse terms
 - $\text{false } 0 \rightarrow \lambda y.y$
 - $\text{if } 0 \text{ then } \dots$
 - 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
 - $\text{false } 0$ // 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

- $e ::= n \mid x \mid \lambda x:t.e \mid e e$
 - We've added integers n as primitives
 - Without at least two distinct types (integer and function), can't have any type errors
 - Functions now include the type of their argument
- $t ::= \text{int} \mid t \rightarrow t$
 - int is the type of integers
 - $t_1 \rightarrow t_2$ is the type of a function that takes arguments of type t_1 and returns a result of type t_2
 - t_1 is the *domain* and t_2 is the *range*
 - Notice this is a recursive definition, so that we can give types to higher-order functions

Looping?

- Simply-typed lambda calculus disallows looping

$D = \lambda x:?. x \ x$

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
- \bullet 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

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

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

$$\frac{A, x:t \vdash e : t'}{A \vdash \lambda x:t.e : t \rightarrow t'} \quad \frac{A \vdash e : t \rightarrow t' \quad A \vdash e' : t}{A \vdash e e' : t'}$$

Example

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

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

$$\frac{\frac{\frac{B \vdash + : i \rightarrow i \rightarrow i \quad B \vdash x : \text{int}}{B \vdash + x : \text{int} \rightarrow \text{int}} \quad B \vdash 3 : \text{int}}{B \vdash + x 3 : \text{int}} \quad A \vdash (\lambda x:\text{int}. + x 3) : \text{int} \rightarrow \text{int} \quad A \vdash 4 : \text{int}}{A \vdash (\lambda x:\text{int}. + x 3) 4 : \text{int}}$$

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 \times expression \rightarrow type

TypeCheck(A, n) = int

TypeCheck(A, x) = if x in dom(A) then A(x) else fail

TypeCheck(A, $\lambda x:t.e$) =

let t' = TypeCheck((A, x:t), e) in t \rightarrow 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