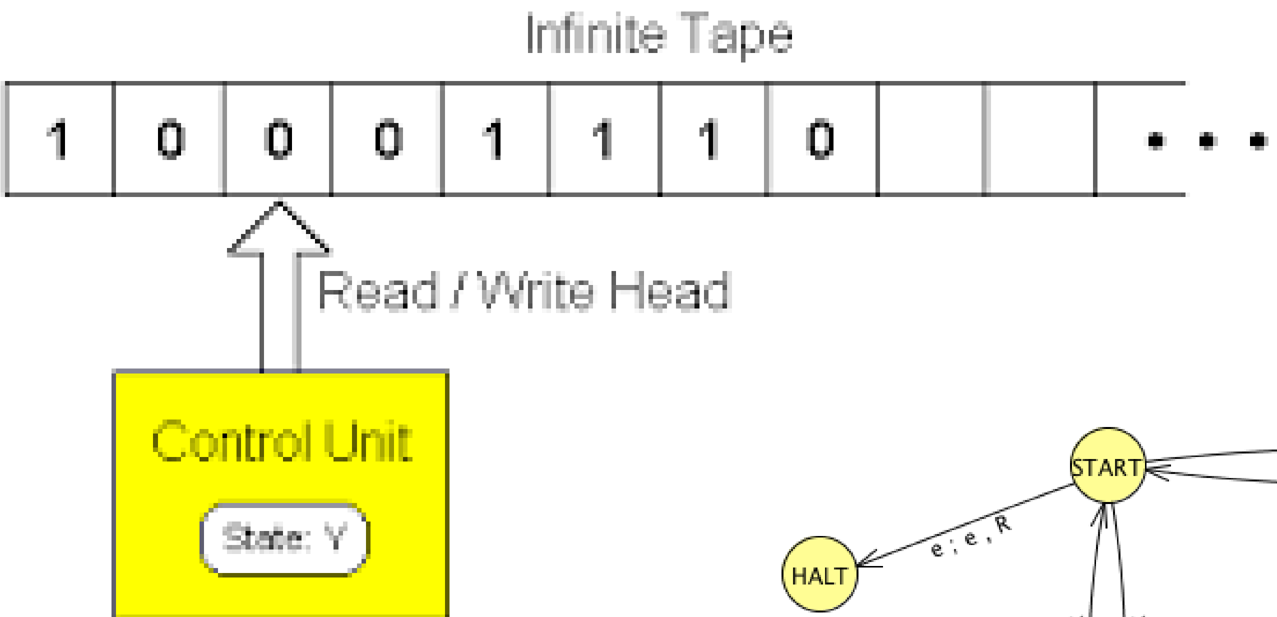
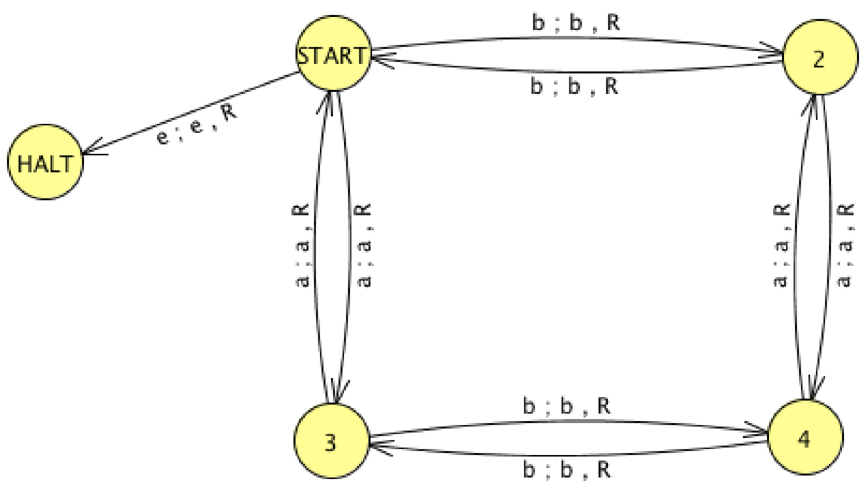
CMSC 330: Organization of Programming Languages

Lambda Calculus

Turing Machine





Lambda Calculus (λ-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washingon DC!)
- Formal system
  - Designed to investigate functions & recursion
  - For exploration of foundations of mathematics
- Now used as
  - Tool for investigating computability
  - Basis of functional programming languages
    - Lisp, Scheme, ML, OCaml, Haskell…

Why Study Lambda Calculus?

- It is a "core" language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, …

Lambda Calculus Syntax

- A lambda calculus expression is defined as

e ::= x	**variable**
| λx.e	**abstraction** (fun def)
| e e	**application** (fun call)

  - λx.e is like `(fun x -> e)` in OCaml

Two Conventions

- Scope of λ extends as far right as possible
  - Subject to scope delimited by parentheses
  - λx. λy.x y is same as λx.(λy.(x y))

- Function application is left-associative
  - x y z is (x y) z
  - Same rule as OCaml

Quiz

This term is equivalent to which of the following?

```
λx.x a b
```

A. `(λx.x) (a b)`
B. `(((λx.x) a) b)`
C. `λx.(x (a b))`
D. `(λx.((x a) b))`

Quiz

This term is equivalent to which of the following?

`λx.x a b`

A. `(λx.x) (a b)`
B. `(((λx.x) a) b)`
C. `λx.(x (a b))`
D. `(λx.((x a) b))`

Lambda Calculus Semantics

- Evaluation: (λx.e1) e2
  - Evaluate e1 with x replaced by e2

- Beta-reduction (*substitution)*

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

Beta Reduction Example

- (λx.λz.x z) y

- Equivalent OCaml code
  - (fun x -> (fun z -> (x z))) y → fun z -> (y z)

Eager Evaluation

- Notice that we evaluated the argument e2 before performing the beta-reduction
  - This is the first version we saw
- Hence, *eager*

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e2 \Downarrow e4 \qquad e3[x:=e4] \Downarrow e5}{e1\ e2 \Downarrow e5}$$

Lazy Evaluation

- Alternatively, we could have performed beta reduction *without* evaluating e2; use it as is
  - Hence, *lazy*

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e3[x:=e2] \Downarrow e4}{e1\ e2 \Downarrow e4}$$

Beta Reductions (CBV)

- $(\lambda x.x)\ z \rightarrow z$
- $(\lambda x.y)\ z \rightarrow y$
- $(\lambda x.x\ y)\ z \rightarrow z\ y$
  - A function that applies its argument to y

Beta Reductions (CBV)

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

Quiz #3

`(λx.y) z` can be beta-reduced to

A. **`y`**
B. **`y z`**
C. **`z`**
D. cannot be reduced

Quiz #3

`(λx.y) z` can be beta-reduced to

A. `y`
B. `y z`
C. `z`
D. cannot be reduced

Quiz #4

Which of the following reduces to λz. z?

a) (λy. λz. x) z
b) (λz. λx. z) y
c) (λy. y) (λx. λz. z) w
d) (λy. λx. z) z (λz. z)

Quiz #4

Which of the following reduces to λz. z?

a) (λy. λz. x) z
b) (λz. λx. z) y
c) (λy. y) (λx. λz. z) w
d) (λy. λx. z) z (λz. z)

CBN Reduction

- CBV
  - (λz.z) ((λy.y) x) → (λz.z) x → x
- CBN
  - (λz.z) ((λy.y) x) → (λy.y) x → x

Beta Reductions (CBN)

(λx.x (λy.y)) (u r) →

(λx.(λw. x w)) (y z) →

Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - 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 (λx.x (λy.y))
  - Renaming bound variables consistently preserves meaning
    - This is called alpha-renaming or alpha conversion
  - Ex. λx.x = λy.y = λz.z   λy.λx.y = λz.λx.z

Quiz #5

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) $\lambda y.\ y\ y$

b) $\lambda z.\ y\ z$

c) $(\lambda x.\ \lambda z.\ x\ z)\ y$

d) $(\lambda x.\ \lambda y.\ x\ y)\ z$

Quiz #5

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) λy. y y

b) λz. y z

c) (λx. λz. x z) y

d) (λx. λy. x y) z

Getting Serious about Substitution

- We have been thinking informally about substitution, but the details matter
- So, let’s carefully formalize it, to help us see where it can get tricky!

Defining Substitution

Substitution: (λx.e1) e2 → e1[x:=e2]

1. (λx.x) e2 → x[x:=e2] = e2 // Replace x by e

Defining Substitution

Substitution: $(\lambda x.e1)\ e2 \rightarrow e1[x:=e2]$

2. $(\lambda x.y)\ e2 \rightarrow y[x:=e2] = y$

y is different than x, so no effect

Defining Substitution

Substitution: (λx.e1) e2 → e1[x:=e2]

3. (λx. e0 e1) e2 → (e0 e1)[x:=e2] →
(e0[x:=e2]) (e1[x:=e2])

Substitute both parts of application

Defining Substitution

Substitution: $(\lambda x.e1)\ e2 \rightarrow e1[x{:=}e2]$

4. $(\lambda x.\ (\lambda x.e'))\ e2 \rightarrow (\lambda x.e')[x{:=}e] \rightarrow \lambda x.e'$

Example:

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

Defining Substitution

Substitution: $(\lambda x.e1)\ e2 \rightarrow e1[x:=e2]$

5. $(\lambda x.\ (\lambda y.e'))\ e2 \rightarrow (\lambda y.e')[x:=e] = ?$

$(\lambda y.(e'[x:=e2]))$ If $y \notin (fvs\ e2)$

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

We want to avoid capturing (free) occurrences of y in e. Change y to a fresh variable w that does not appear in e' or e

$(\lambda y.(e'[x:=e2]))$ alpha-convert e' if $y \in (fvs\ e2)$

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

- Formally:

  $(\lambda y.e')[x:=e] = \lambda w.((e'\ [y:=w])\ [x:=e])$ (w is fresh)

Free Variables

$FV(x) = \{x\}$
$FV\ (e1\ e2) = FV\ (e1) \cup FV\ (e2)$
$FV\ (\lambda x.e) = FV(e) - \{x\}$

Example:
$FV(x) = \{x\}$
$FV(x\ y) = \{x,y\}$
$FV(\lambda x.\ x) = FV(x) - \{x\} = \{\ \}$
$FV(\lambda x.\ x\ y) = FV(x\ y) – \{x\} = \{y\}$
$FV((\lambda x.\ x\ y)\ x) = FV(\lambda x.\ x\ y) \cup FV(x) = \{x,y\}$

Lambda Calc, Impl in OCaml

- e ::= x
  | λx.e
  | e e

```
type id = string
type exp = Var of id
| Lam of id * exp
| App of exp * exp
```

y
```
Var "y"
```

λx.x
```
Lam ("x", Var "x")
```

λx.λy.x y
```
Lam ("x",(Lam("y",App (Var "x", Var "y"))))
```

(λx.λy.x y) λx.x x
```
App
  (Lam("x",Lam("y",App(Var"x",Var"y"))),
   Lam ("x", App (Var "x", Var "x")))
```

OCaml Implementation: Substitution

```
(* substitute e for y in m--   m[y:=e]     *)
let rec subst m y e =
  match m with
      Var x ->
        if y = x then e (* substitute *)
        else m          (* don't subst *)
    | App (e1,e2) ->
        App (subst e1 y e, subst e2 y e)
    | Lam (x,e0) -> …
```

OCaml Impl: Substitution (cont'd)

```
(* substitute e for y in m--   m[y:=e]     *)
let rec subst m y e = match m with …
    | Lam (x,e0) ->
      if y = x then m
      else if not (List.mem x (fvs e)) then
        Lam (x, subst e0 y e)
      else
        let z = newvar() in (* fresh *)
        let e0' = subst e0 x (Var z) in
        Lam (z,subst e0' y e)
```

Shadowing blocks substitution

Safe: no capture possible

Might capture; need to α-convert

CBV, L-to-R Reduction with Partial Eval

```
let rec reduce e =
  match e with
      App (Lam (x,e), e2) -> subst e x e2
    | App (e1,e2) ->
      let e1' = reduce e1 in
      if e1' != e1 then App(e1',e2)
      else App (e1,reduce e2)
    | Lam (x,e) -> Lam (x, reduce e)
    | _ -> e
```

Straight β rule

Reduce lhs of app

Reduce rhs of app

Reduce function body

nothing to do

The Power of Lambdas

- To give a sense of how one can encode various constructs into LC we'll be looking at some concrete examples:
  - Let bindings
  - Booleans
  - Pairs
  - Natural numbers & arithmetic
  - Looping

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) → (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) → 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

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)

Operations On Church Numerals

- Successor
  - succ = $\lambda z.\lambda f.\lambda y.f\ (z\ f\ y)$

- $0 = \lambda f.\lambda y.y$
- $1 = \lambda f.\lambda y.f\ y$

- Example
  - succ 0 =

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

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

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

    $\lambda f.\lambda y.f\ y$

    $= 1$

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

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 →

    true

  - 0 = λf.λy.y

Since (λx.y) z → 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)

Arithmetic (cont.)

- $1 = \lambda f.\lambda y.f\ y$
- $2 = \lambda f.\lambda y.f\ (f\ y)$

- Prove 1+1 = 2
  - $1+1 = \lambda x.\lambda y.(1\ x)\ (1\ x\ y) =$
  - $\lambda x.\lambda y.((\lambda f.\lambda y.f\ y)\ x)\ (1\ x\ y) \rightarrow$
  - $\lambda x.\lambda y.(\lambda y.x\ y)\ (1\ x\ y) \rightarrow$
  - $\lambda x.\lambda y.x\ (1\ x\ y) \rightarrow$
  - $\lambda x.\lambda y.x\ ((\lambda f.\lambda y.f\ y)\ x\ y) \rightarrow$
  - $\lambda x.\lambda y.x\ ((\lambda y.x\ y)\ y) \rightarrow$
  - $\lambda x.\lambda y.x\ (x\ y) = 2$
- With these definitions
  - Can build a theory of arithmetic

Arithmetic Using Church Numerals

- What about subtraction?
  - Easy once you have 'predecessor', but...
  - Predecessor is very difficult!
- Story time:
  - One of Church's students, Kleene (of Kleene-star fame) was struggling to think of how to encode 'predecessor', until it came to him during a trip to the dentists office.
  - Take from this what you will
- Wikipedia has a great derivation of 'predecessor'.

Looping+Recursion

- So far we have avoided self-reference, so how does recursion work?
- We can construct a lambda term that ‘replicates’ itself:
  - Define $D = \lambda x.x\ x$, then
    - $D\ D = (\lambda x.x\ x)\ (\lambda x.x\ x) \rightarrow (\lambda x.x\ x)\ (\lambda x.x\ x) = D\ D$
  - $D\ D$ is an infinite loop
- We want to generalize this, so that we can make use of 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

Factorial 4=?

```
(Y G) 4
 G (Y G) 4
(λr.λn.(if n = 0 then 1  else n × (r (n−1)))) (Y G) 4
(λn.(if n = 0 then 1 else n × ((Y G) (n−1)))) 4
if 4 = 0 then 1 else 4 × ((Y G) (4−1))
4 × (G (Y G) (4−1))
4 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (4−1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3−1)))
4 × (3 × (G (Y G) (3−1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (3−1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2−1))))
4 × (3 × (2 × (G (Y G) (2−1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (2−1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1−1)))))
4 × (3 × (2 × (1 × (G (Y G) (1−1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (1−1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0−1))))))
4 × (3 × (2 × (1 × (1))))
24
```

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, at least not as written earlier.
- 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