CMSC 330: Organization of Programming Languages

Lambda Calculus
Turing Machine

Infinite Tape

1 0 0 0 1 1 1 0 ...

Read / Write Head

Control Unit
State: Y

START

HALT

e; e, R

b; b, R

a; a, R

a; a, R

b; b, R

b; b, R

3

4

2
A language $L$ is Turing complete if it can compute any function computable by a Turing Machine.

Show a language $L$ is Turing complete if

- We can map every Turing machine to a program in $L$.
  - I.e., a program can be written to emulate a Turing machine.
- Or, we can map any program in a known Turing-complete language to a program in $L$.

Turing complete languages the “most powerful”

- *Church-Turing thesis (1936)*: Computability by a Turing Machine defines “effectively computable”
Programming Language Expressiveness

- So what language features are needed to express all computable functions?
  - What’s a minimal language that is Turing Complete?

- Observe: some features exist just for convenience
  - Multi-argument functions `foo ( a, b, c )`
    - Use currying or tuples
  - Loops `while (a < b) ...`
    - Use recursion
  - Side effects `a := 1`
    - Use functional programming pass “heap” as an argument to each function, return it when with function’s result
Lambda Calculus ($\lambda$-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washington 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…
Lambda Calculus Syntax

- A lambda calculus expression is defined as

\[ e ::= x \quad \text{variable} \]
\[ \mid \lambda x.e \quad \text{abstraction (func def)} \]
\[ \mid e e \quad \text{application (func call)} \]

- This grammar describes ASTs; not for parsing (ambiguous!)
- Lambda expressions also known as lambda terms

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

That's it! Nothing but (higher-order) functions
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, …
- Plus, higher-order, anonymous functions (aka *lambdas*) are now very popular!
  - C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), … (and functional languages like OCaml, Haskell, F#, …)
Two Conventions

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

- Function application is left-associative
  - \( x \ y \ z \) is \((x \ y) \ z \)
  - Same rule as OCaml
OCaml Lambda Calc Interpreter

\[ e ::= x \]
\[ | \lambda x.e \]
\[ | e e \]

\[ y \]
\[ \lambda x.x \]
\[ \lambda x.\lambda y.x y \]
\[ (\lambda x.\lambda y.x y) \lambda x.x \]

\[ \text{type id} = \text{string} \]
\[ \text{type exp} = \text{Var of id} \]
\[ | \text{Lam of id * exp} \]
\[ | \text{App of exp * exp} \]

\[ \text{Var "y"} \]
\[ \text{Lam ("x", Var "x")} \]
\[ \text{Lam ("x", (Lam("y", App (Var "x", Var "y")))}) \]
\[ \text{App (Lam("x", Lam("y", App (Var "x", Var "y"))))}, \]
\[ \text{Lam ("x", App (Var "x", Var "x"))} \]
Quiz #1

\( \lambda x. (y \ z) \) and \( \lambda x. y \ z \) are equivalent

A. True
B. False
Quiz #1

$\lambda x. (y \ z)$ and $\lambda x. y \ z$ are equivalent

A. True
B. False
Quiz #2

What is this term’s AST?

\[ \lambda x . x x \]

A. App (Lam ("x", Var "x"), Var "x")
B. Lam (Var "x", Var "x", Var "x")
C. Lam ("x", App (Var "x", Var "x"))
D. App (Lam ("x", App ("x", "x")))
Quiz #2

What is this term’s AST?

\[ \lambda x.x \ x \ x \]

A. App (Lam ("x", Var "x"), Var "x")
B. Lam (Var "x", Var "x", Var "x")
C. Lam ("x", App (Var "x", Var "x"))
D. App (Lam ("x", App ("x", "x")))

type id = string

\[ \text{type exp =} \]

Var of id
| Lam of id * exp
| App of exp * exp
Quiz #3

This term is equivalent to which of the following?

\[ \lambda x. x \ a \ b \]

A. \((\lambda x. x) \ (a \ b)\)
B. \(((\lambda x. x) \ a) \ b)\)
C. \(\lambda x. (x \ (a \ b))\)
D. \((\lambda x. ((x \ a) \ b))\)
Quiz #3

This term is equivalent to which of the following?

\( \lambda x. x \ a \ b \)

A. \((\lambda x. x) \ (a \ b)\)
B. \(((\lambda x. x) \ a) \ b)\)
C. \(\lambda x. (x \ (a \ b))\)
D. \((\lambda x. (((x \ a) \ b))\)
Lambda Calculus Semantics

- Evaluation: All that’s involved are function calls 
  \((\lambda x.e_1) \; e_2\)
  - Evaluate \(e_1\) with \(x\) replaced by \(e_2\)

- This application is called **beta reduction**
  - \((\lambda x.e_1) \; e_2 \rightarrow e_1\{e_2/x\}\)
    - \(e_1\{e_2/x\}\) is \(e_1\) with occurrences of \(x\) replaced by \(e_2\)
    - This operation is called **substitution**
      - Replace formal parameters with actual arguments
      - Instead of using environment to map formals to actuals
    - We allow reductions to occur **anywhere** in a term
      - Order reductions are applied does not affect final value!

- When a term **cannot be reduced further** it is in **beta normal form**
Beta Reduction Example

1. \((\lambda x.\lambda z.x \, z) \, y\)
   \[\rightarrow (\lambda x.(\lambda z.(x \, z))) \, y\] // since \(\lambda\) extends to right
   \[\rightarrow (\lambda x.(\lambda z.(x \, z))) \, y\] // apply \((\lambda x.e1) \, e2 \rightarrow e1\{e2/x\}\)
   // where \(e1 = \lambda z.(x \, z), \, e2 = y\)
   \[\rightarrow \lambda z.(y \, z)\] // final result

2. Equivalent OCaml code
   
   - \((\text{fun } x \rightarrow (\text{fun } z \rightarrow (x \, z))) \, y \rightarrow \text{fun } z \rightarrow (y \, z)\)
Beta Reduction Examples

1. \((\lambda x.x) \ z \rightarrow z\)
2. \((\lambda x.y) \ z \rightarrow y\)
3. \((\lambda x.x \ y) \ z \rightarrow z \ y\)
   - A function that applies its argument to \(y\)
Beta Reduction Examples (cont.)

- \((\lambda x. x \ y) \ (\lambda z. z) \rightarrow (\lambda z. z) \ y \rightarrow y\)

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

- \((\lambda x. \lambda y. x \ y) \ (\lambda z. z z) \ x \rightarrow (\lambda y. (\lambda z. z z) y) x \rightarrow (\lambda z. z z) x \rightarrow x x\)
Beta Reduction Examples (cont.)

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

$$(\lambda x.(\lambda w. x \ w)) \ (\lambda z.z) \rightarrow (\lambda w. (\lambda z.z) \ w) \rightarrow (\lambda w.w)$$
Quiz #4

\((\lambda x. y) \; z\) can be beta-reduced to

A. \(y\)
B. \(y \; z\)
C. \(z\)
D. cannot be reduced
Quiz #4

\((\lambda x. y) \; z\) can be beta-reduced to

A. \(y\)
B. \(y \; z\)
C. \(z\)
D. cannot be reduced
Quiz #5

Which of the following reduces to $\lambda z. z$?

a) $(\lambda y. \lambda z. x) z$
b) $(\lambda z. \lambda x. z) y$
c) $(\lambda y. y) (\lambda x. \lambda z. z) w$
d) $(\lambda y. \lambda x. z) z (\lambda z. z)$
Quiz #5

Which of the following reduces to $\lambda z \cdot z$?

a) $(\lambda y \cdot \lambda z \cdot x) \cdot z$

b) $(\lambda z \cdot \lambda x \cdot z) \cdot y$

c) $(\lambda y \cdot y) (\lambda x \cdot \lambda z \cdot z) \cdot w$

d) $(\lambda y \cdot \lambda x \cdot z) \cdot z (\lambda z \cdot z)$
Static Scoping & 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 preserves meaning
    - This is called alpha-renaming or alpha conversion
  - Ex. \(\lambda x.x = \lambda y.y = \lambda z.z\)  \(\lambda y.\lambda x.y = \lambda z.\lambda x.z\)
Terminology: Free and Bound Variables

- **A free variable** is one that doesn’t have a surrounding lambda that binds it
  - In \((\lambda y. y \ z \ x)\), the variables \(z\) and \(x\) are free
  - In \((\lambda y. \lambda z. y \ z \ x)\), the variable \(x\) is free
  - In \((\lambda y. \lambda z. y \ z)\), there are no free variables

- **A bound variable** is one that does have a corresponding binder
  - In \((\lambda y. y \ z \ x)\), the variable \(y\) is bound (but not \(z\) and \(x\))
  - In \((\lambda y. \lambda z. y \ z \ x)\), the variables \(y\) and \(z\) are bound (not \(x\))
  - In \((\lambda y. \lambda z. y)\), the variable \(y\) is bound (\(z\) does not appear)
Quiz #6

Which of the following expressions is alpha equivalent to \((\lambda x. \lambda y. x y) y\)

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 #6

Which of the following expressions is alpha equivalent to \((\lambda x. \, \lambda y. \, x \, y) \, y\)

a) \(\lambda y. \, y \, y\)
b) \(\lambda z. \, y \, z\)
c) \((\lambda x. \, \lambda z. \, x \, z) \, y\)
d) \((\lambda x. \, \lambda y. \, x \, y) \, z\)
Defining Substitution

- Use recursion on structure of terms
  - $x\{e/x\} = e$  // Replace $x$ by $e$
  - $y\{e/x\} = y$  // $y$ is different than $x$, so no effect
  - $(e_1 \ e_2)\{e/x\} = (e_1\{e/x\}) \ (e_2\{e/x\})$
    // Substitute both parts of application
  - $(\lambda x.e)\{e/x\} = \lambda x.e'$
    - In $\lambda x.e'$, the $x$ is a parameter, and thus a local variable that is different from other $x$'s. Implements static scoping.
    - So the substitution has no effect in this case, since the $x$ being substituted for is different from the parameter $x$ that is in $e'$
  - $(\lambda y.e')\{e/x\} = ?$
    - The parameter $y$ does not share the same name as $x$, the variable being substituted for
    - Is $\lambda y. (e' \ {e/x})$ correct? No…
Variable capture

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 \(y\), as this violates static scoping
- I.e., \((\lambda x. \lambda y. x \ y) \ y \neq \lambda y.y \ y\)

Solution

- \((\lambda x. \lambda y. x \ y)\) is “the same” as \((\lambda x. \lambda z. x \ z)\)
  - Due to alpha conversion
- So alpha-convert \((\lambda x. \lambda y. x \ y) \ y\) to \((\lambda x. \lambda z. x \ z) \ y\) first
  - Now \((\lambda x. \lambda z. x \ z) \ y \rightarrow \lambda z.y \ z\)
Completing the Definition of Substitution

- Recall: we need to define \((\lambda y. e')\{e/x}\)
  - We want to avoid capturing \textbf{free} occurrences of \(y\) in \(e\)
  - Solution: alpha-conversion!
    - Change \(y\) to a variable \(w\) that does not appear in \(e'\) or \(e\)
      (Such a \(w\) is called \textit{fresh})
    - Replace all occurrences of \(y\) in \(e'\) by \(w\).
    - Then replace all occurrences of \(x\) in \(e'\) by \(e\!\)

- Formally:
  \[
  (\lambda y. e')\{e/x\} = \lambda w. (e'\{w/y\})\{e/x\} \quad (w \text{ is fresh WRT } e \text{ and } e')
  \]
Beta-Reduction, Again

Whenever we do a step of beta reduction

- \((\lambda x. e_1) e_2 \rightarrow e_1[e_2/x]\)
- We alpha-convert variables as necessary
- Sometimes performed implicitly (w/o showing conversion)

Examples

- \((\lambda x. \lambda y. x \ y) \ y = (\lambda x. \lambda z. x \ z) \ y \rightarrow \lambda z.y \ z \quad // \ y \rightarrow \ z\)
- \((\lambda x.x \ (\lambda x.x)) \ z = (\lambda y.y \ (\lambda x.x)) \ z \rightarrow z \ (\lambda x.x) \quad // \ x \rightarrow \ y\)
OCaml Implementation: Free variables

(* compute free variables in e *)

let rec fvs e =
    match e with
    | Var x -> [x]  (* "Naked" variable is free *)
    | App (e1,e2) -> (fvs e1) @ (fvs e2)
    | Lam (x,e0) -> List.filter (fun y -> x <> y) (fvs e0)
        (* Append free vars of sub-expressions *)
        (* Filter x from the free variables in e0 *)
OCaml Implementation: Substitution

(* substitute e for y in m-- m{e/y} *)
let rec subst e y m =
  match m with
  | Var x ->
    if y = x then e (* substitute *)
    else m (* don’t subst *)
  | App (e1,e2) ->
    App (subst e y e1, subst e y e2)
  | Lam (x,e0) -> ...

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

let rec reduce e =
  match e with
  | App (Lam (x,e), e2) -> subst e2 x e
  | 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
Quiz #7

Beta-reducing the following term produces what result?

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

A. $y \ (\lambda z. z \ y)$
B. $z \ (\lambda y. y \ z)$
C. $y \ (\lambda y. y \ y)$
D. $y \ y$
Quiz #7

Beta-reducing the following term produces what result?

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

A. $y \ (\lambda z.z \ y)$
B. $z \ (\lambda y.y \ z)$
C. $y \ (\lambda y.y \ y)$
D. $y \ y$
Quiz #8

Beta reducing the following term produces what result?

\[ \lambda x. (\lambda y. y y) w z \]

a) \( \lambda x. w w z \)
b) \( \lambda x. w z \)
c) \( w z \)
d) Does not reduce
Quiz #8

Beta reducing the following term produces what result?

\[ \lambda x. (\lambda y. y y) \text{ w z} \]

a) \( \lambda x. \text{ w w z} \)
b) \( \lambda x. \text{ w z} \)
c) \( \text{w z} \)
d) Does not reduce