CMSC 330: Organization of Programming Languages

Lambda Calculus
Is there an algorithm to determine if a statement is true in all models of a theory?
Entscheidungsproblem “decision problem“

Algorithm, formalised

Alonzo Church: Lambda calculus
An unsolvable problem of elementary number theory, *Bulletin the American Mathematical Society*, May 1935

Kurt Gödel: Recursive functions

Alan M. Turing: Turing machines
On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, received 25 May 1936
Turing Machine

Infinite Tape

1 0 0 0 1 1 1 0 ...

Control Unit
State: Y

Read / Write Head

HALT

START

3

4

b; b, R

a; a, R

a; a, R

b; b, R

b; b, R

b; b, R

START

3

4

b; b, R

a; a, R

a; a, R

b; b, R

b; b, R

b; b, R

a; a, R

a; a, R

b; b, R

b; b, R

a; a, R

a; a, R

b; b, R

b; b, R

a; a, R

a; a, R

b; b, R

b; b, R

a; a, R

a; a, R

b; b, R

b; b, R
Turing Completeness

- Turing machines are the most powerful description of computation possible
  - They define the Turing-computable functions
- 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
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:
  - Use currying or tuples
  ```
  foo ( a, b, c )
  ```

• Loops:
  - Use recursion
  ```
  while (a < b) …
  ```

• Side effects:
  - Use functional programming pass “heap” as an argument to each function, return it when with function’s result
  ```
  a := 1
  ```
Mini C

You only have:
• If statement
• Plus 1
• Minus 1
• functions

Sum \( n = 1+2+3+4+5\ldots n \) in Mini C

```c
int add1(int n){return n+1;}
int sub1(int n){return n-1;}
int add(int a,int b){
    if(b == 0) return a;
    else return add( add1(a),sub1(b));
}
int sum(int n){
    if(n == 1) return 1;
    else return add(n, sum(sub1(n)));
}
int main(){
    printf("%d\n",sum(5));
}
```
Lambda Calculus (λ-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

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

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

- \( \lambda x.e \) is like \( \text{(fun } x \rightarrow e) \) 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#, …)
Three 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

- As a convenience, we use the following “syntactic sugar” for local declarations
  - $\text{let } x = e_1 \text{ in } e_2$ is short for $(\lambda x.e_2)\ e_1$
OCaml Lambda Calc Interpreter

- \( e ::= x \)
- \( \lambda x. e \)
- \( e e \)

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

- \( y \) \quad \text{Var } "y" \\
- \( \lambda x. x \) \quad \text{Lam ("x", Var "x")} \\
- \( \lambda x. \lambda y. x \ y \) \quad \text{Lam ("x", (Lam("y", App (Var "x", Var "y")))))} \\
- \( (\lambda x. \lambda y. x \ y) \ \lambda x. x \ x \) \quad \text{App (Lam("x",Lam("y",App (Var"x",Var"y"))), 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

\[ \lambda x. (y \ z) \text{ and } \lambda x. y \ z \text{ are equivalent} \]

A. True
B. False
Quiz #2

What is this term’s AST?

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

A. \( \text{App} (\text{Lam} ("x", \text{Var} "x"), \text{Var} "x") \)
B. \( \text{Lam} (\text{Var} "x", \text{Var} "x", \text{Var} "x") \)
C. \( \text{Lam} ("x", \text{App} (\text{Var} "x", \text{Var} "x")) \)
D. \( \text{App} (\text{Lam} ("x", \text{App} ("x", "x"))) \)

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

```plaintext
type id = string
type exp =
    Var of id
| Lam of id * exp
| App of exp * exp
```
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. \((\lambda x. ((x a) b))\)
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**
  - $(\lambda x. e_1) \ e_2 \rightarrow e_1[x:=e_2]$
    - $e_1[x:=e_2]$ is $e_1$ with occurrences of $x$ replaced by $e_2$
    - This operation is called **substitution**
      - Replace formals with actuals
      - 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

- \((\lambda x.\lambda z. x\ z)\ y\)
  - \(\rightarrow (\lambda x. (\lambda z. (x\ z)))\ y\) // since \(\lambda\) extends to right
  - \(\rightarrow (\lambda x. (\lambda z. (x\ z)))\ y\) // apply \((\lambda x. e_1)\ e_2 \rightarrow e_1[x:=e_2]\)
  - // where \(e_1 = \lambda z. (x\ z)\), \(e_2 = y\)
  - \(\rightarrow \lambda z. (y\ z)\) // final result

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

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

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

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

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

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

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

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

A. $y$
B. $y z$
C. $z$
D. cannot be reduced
Quiz #4

(λx. 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 $\lambda z. z$?

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) $(\lambda y. \lambda x. z) z (\lambda z. z)$
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. \(\lambda x.x = \lambda y.y = \lambda z.z\) \quad \lambda y.\lambda x.y = \lambda z.\lambda x.z\)
Quiz #6

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 #6

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
Defining Substitution

- Use recursion on structure of terms
  - $x[x:=e] = e$  // Replace $x$ by $e$
  - $y[x:=e] = y$  // $y$ is different than $x$, so no effect
  - $(e_1 e_2)[x:=e] = (e_1[x:=e]) (e_2[x:=e])$
    // Substitute both parts of application
  - $(\lambda x.e')[x:=e] = \lambda x.e'$
    - 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 $e'$
  - $(\lambda y.e')[x:=e] = ?$
    - The parameter $y$ does not share the same name as $x$, the variable being substituted for
    - Is $\lambda y.(e'[x:=e])$ 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 \((\lambda y.e')[x:=e]\)
  - We want to avoid capturing (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 fresh)
    - Replace all occurrences of \(y\) in \(e'\) by \(w\).
    - Then replace all occurrences of \(x\) in \(e'\) by \(e\).

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

Whenever we do a step of beta reduction

- \((\lambda x. e_1) e_2 \rightarrow e_1[x:=e_2]\)
- We must 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: 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) -> ...
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  Might capture; need to α-convert

let z = newvar() in (* fresh *)
let e0' = subst e0 x (Var z) in 
Lam (z,subst e0' y e)
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

nothing to do
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?

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

a) \( \lambda x. \text{ w w z} \)

b) \( \lambda x. \text{ w z} \)

c) \( \text{ w z} \)

 d) Does not reduce
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