Foundations of Theory of Programming Languages: Introduction to Lambda Calculus

Lecture 23

Lecture Outline

- Some History
- Why study lambda calculus?
- What IS lambda calculus?
- How extensions relate to explaining Lisp, Tiger, Java etc. semantics

Lambda Calculus. History.

- A framework developed in 1930s by Alonzo Church to study computations with functions
- Church wanted a minimal notation
  - to expose only what is essential
- Two operations with functions are essential:
  - function creation
  - function application
- Um, what has this to do with (integral?) calculus?

Function Creation

- Church introduced the notation

  $$\lambda x.\ E$$

  to denote a function with formal argument x and with body E
- Functions do not have names
  - names are not essential for the computation
- Functions have a single argument
  - once we understand how functions with one argument work we can generalize to multiple args.

History of Notation

- Whitehead & Russell (*Principia Mathematica*) used the notation $\hat{y}$ P to denote the set of y's such that P holds.
- Church borrowed the notation but moved ^ down to create $\wedge y$ E
- Which later turned into λy. E and the calculus became known as lambda calculus
- John McCarthy, inventor of Lisp, who later admitted he didn't really "understand" lambda calculus at the time, appropriated the notation lambda[y](E), later changed to (lambda(y) E) for an "anonymous" function.

More on Lisp and Lambda...

- We will find it useful to use lisp notation in these slides because logicians muck up their formal notation with implicit operators (just putting items next to each other) and expecting the reader to figure out precedence.

Preview: Function Application

- The **only** thing that we can do with a function is to apply it to an argument
- Church used the notation

  $E_1\ E_2$ …in lisp, $(E_1\ E_2\ )$

  to denote the application of function E_1 to actual argument E_2
- All functions are applied to a single argument
- Even so, THIS IS ENOUGH TO COMPUTE ANYTHING.

It sounds too dumb!! Why Study Lambda Calculus?

- λ-calculus has had a tremendous influence on the theory and analysis of programming languages
- Comparable to Turing machines
- Provides a PL framework:
  - Realistic languages are too large and complex to study from scratch as a whole
  - Typical approach is to modularize the study into one feature at a time
    - E.g., recursion, looping, exceptions, objects, etc.
  - Then we assemble the features together

Is lambda calculus a programming language?

- λ-calculus as a concept is the standard testbed for studying programming language features
  - Because of its minimality
  - Despite its syntactic simplicity the λ-calculus can easily encode:
    - numbers, recursive data types, modules, imperative features, exceptions, etc.
- Certain language features necessitate more substantial extensions to λ-calculus:
  - for distributed & parallel languages: π-calculus
  - for object oriented languages: σ-calculus
- But as will be evident shortly, the bare lambda calculus is not a PL in a practical sense.

A prediction come true

“*Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.*"

(Peter Landin 1966)

Syntax of Lambda Calculus

- Syntax: Only three kinds of expressions

$E \rightarrow x$	variables
$\mid E_1\ E_2$	function application
$\mid \lambda x.\ E$	function creation

- The form $\lambda x.\ E$ is also called lambda abstraction, or simply abstraction
- E are called λ-terms or λ-expressions

Syntax of Lambda Calculus in Lisp

- Only three kinds of LISP expressions

$E \to x$	variables
$\mid$ $(E_1\ E_2)$	function application
$\mid$ (lambda(x) E)	function creation

- The form (lambda(x)E) is also called lambda abstraction, or simply <u>abstraction</u>
- E are called s-expressions or terms

Examples of Lambda Expressions

- The identity function:

  $I =_{def} \lambda x.\ x$ … (lambda(x) x)

- A function that given an argument y discards it and computes the identity function:

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

- A function that given a function f invokes it on the identity function

  $\lambda f.\ f\ (\lambda\ x.\ x)$ …(lambda(f)(f(lambda(x)x)))

  {actually, that's Scheme. In CL we need…(lambda(f)(funcall f (lambda(x)x))) }

Notational Conventions which primarily serve to confuse.

- Application associates to the left

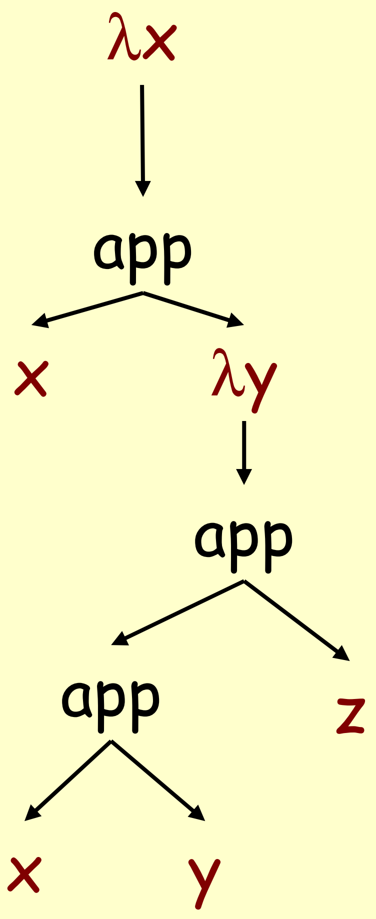
  x y z parses as (x y) z

- Abstraction extends to the right as far as possible

  λx. x λy. x y z parses as

  λ x. (x (λy. ((x y) z)))

- And yields the the parse tree:



Notational Conventions in Lisp require no precedence to parse

- Application ((x y) z)
- "Abstraction" is also obvious in Lisp syntax

  λx. x λy. x y z parses as

  (lambda(x) (x (lambda(y)((x y) z))

- Note that in this bottom example, x is a function applied to y. x is also a function applied to (lambda(y)((x y) z))

Scope of Variables

- As in all languages with variables it is important to discuss the notion of scope
  - Recall: the scope of an identifier is the portion of a program where the identifier is accessible
- An abstraction $\lambda x.\ E$ binds variable x in E
  - x is the newly introduced variable
  - E is the scope of x
  - we say x is bound in $\lambda x.\ E$ … (lambda(x)E)

Free and Bound Variables

- A variable is said to be free in E if it is not bound in E
- We can define the free variables of an expression E recursively as follows:

  $\mathrm{Free}(x) = \{x\}$

  $\mathrm{Free}(E_1\ E_2) = \mathrm{Free}(E_1) \cup \mathrm{Free}(E_2)$

  $\mathrm{Free}(\lambda x.\ E) = \mathrm{Free}(E) - \{\ x\ \}$
- Example: $\mathrm{Free}(\lambda x.\ x\ (\lambda y.\ x\ y\ z)) = \{\ z\ \}$
- You could think that free variables are global or bound "outside" the expression we are looking at.

Free and Bound Variables Lisp notation

- A variable is said to be <u>free</u> in E if it is not bound in E
- We can define the free variables of an expression E recursively as follows:

  Free(x) = {x}

  Free($(E_1\ E_2)$) = Free(E_1) ∪ Free(E_2)

  Free((lambda(x) E)) = Free(E) - { x }
- Example: Free((lambda(x)(x (lambda(y)((x y) z))))) = { z }
- Lisp would call these global or special vars

Free and Bound Variables Lisp notation

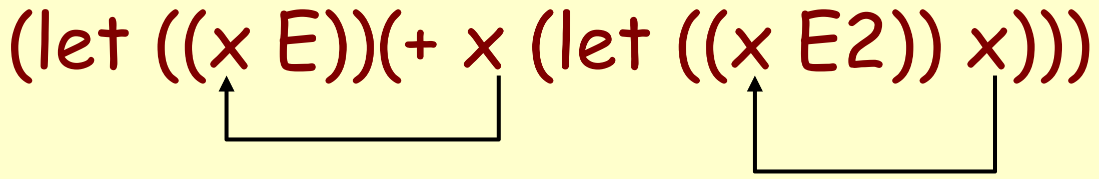
Actually, common lisp is not so functionally pure; Scheme indeed looks more like what we have just written, but in CL we have to use a few extra marks to denote functions.

instead of (lambda(x)(x (lambda(y)((x y) z)…

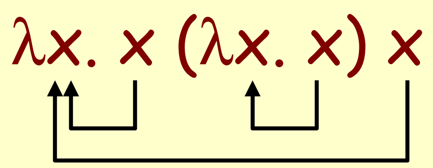
```
(compile nil '(lambda(x)(funcall x #'(lambda(y)(funcall
(funcall x y) z))))

; While compiling (:internal (:anonymous-lambda 0) 0):

Warning: Free reference to undeclared variable z assumed
special.
```

Free and Bound Variables (Cont.)

- Just like in any language with static nested scoping we have to worry about variable shadowing
  - An occurrence of a variable might refer to different things in different context
- E.g. Lisp (let ((x E))(+ x (let ((x E2)) x)))



- In λ-calculus: λx. x (λx. x) x *note we can't do + here*



Renaming Bound Variables

- What if two λ-terms can be obtained from each other by a renaming of the bound variables ?
- Example: $\lambda x.\ x$ is identical to $\lambda y.\ y$ and to $\lambda z.\ z$
- Intuition:
  - by changing the name of a formal argument and of all its occurrences in the function body, the *behavior* of the function cannot change
  - in λ-calculus such functions are considered **identical**
  - In Lisp (lambda(x) x) and (lambda(y) y) are different programs, but computationally equivalent. Mathematicians don't have such subtle notions of difference as programmers…

Renaming Bound Variables (Cont.)

- Convention: we will always rename bound variables so that they are all unique
  - e.g., write $\lambda x.\ x\ (\lambda y.y)\ x$ instead of $\lambda x.\ x\ (\lambda x.x)\ x$
- This makes it easy to see the scope of bindings
- And also prevents serious confusion !
- Bad: (lambda(x)((x (lambda(x)x) x)
- OK: (lambda(x)((x (lambda(y)y) x) well, about as good as it is going to get...

What can we do with lambda calculus expressions?

- Need operations to apply functions taking into account bindings
- If possible, find canonical "simplest" forms for expressions

Substitution

- The substitution of E' for x in E (written [E'/x]E)
  - Step 1. Rename bound variables in E and E' so they are unique
  - Step 2. Perform the textual substitution of E' for x in E
- Example: [y (λx. x) / x] λy. (λx. x) y x
  - After renaming: [y (λv. v)/x] λz. (λu. u) z x
  - After substitution: λz. (λu. u) z (y (λv. v))

  - [(y (lambda(x)x)) / x] (lambda(y)(((lambda(x)x)y)x)
  - [(y (lambda(v)v)) / x] (lambda(z)(((lambda(u)u)z)x)
  - (lambda(z)(((lambda(u)u)y) (y (lambda(v)v)))

Evaluation of λ-terms

- There is one key evaluation step in λ-calculus: the function application

  (λx. E) E′ evaluates to [E′/x]E
- **This is called β-reduction**
- We write $E \to_\beta E'$ to say that E′ is obtained from E in one β-reduction step
- We write $E \to^*_\beta E'$ if there are zero or more steps

Examples of Evaluation

- The identity function:

$$(\lambda x.\ x)\ E \rightarrow [E\ /\ x]\ x = E$$

- Another example with the identity:

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

$[\lambda x.\ x\ /\ f]\ f\ (\lambda x.\ x)) = [(\lambda x.\ x)\ /\ f]\ f\ (\lambda y.\ y)) =$

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

$[\lambda y.\ y\ /x]\ x = \lambda y.\ y$

- A non-terminating evaluation:

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

$[\lambda y.\ yy\ /\ x]xx = (\lambda y.\ yy)(\lambda y.\ yy) \rightarrow \ldots$

Examples of Evaluation in Lisp notation

- The identity function: ((lambda(x)x) E) =E

  $(\lambda x.\ x)\ E \rightarrow [E\ /\ x]\ x = E$

- Another example with the identity:

  ((lambda(f)(f(lambda(x) x))(lambda(x)x)) = … (lambda(y) y)

- A non-terminating evaluation:

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

  $[\lambda y.\ yy\ /\ x]xx = (\lambda y.\ yy)(\lambda y.\ yy) \rightarrow \ldots$

  ((lambda(x)(x x)) (lambda(y) (y y))) beta reduces to

  ((lambda(y)(y y)) (lambda(y) (y y)))

Functions with Multiple Arguments

- Consider that we extend the calculus with the add primitive operation
- The λ-term $\lambda x.\ \lambda y.$ add x y can be used to add two arguments E_1 and E_2:

  $(\lambda x.\ \lambda y.\ \text{add}\ x\ y)\ E_1\ E_2 \rightarrow_\beta$

  $([E_1/x]\ \lambda y.\ \text{add}\ x\ y)\ E_2 =$

  $(\lambda y.\ \text{add}\ E_1\ y)\ E_2 \rightarrow_\beta$

  $[E_2/y]\ \text{add}\ E_1\ y\ = \text{add}\ E_1\ E_2$
- The arguments are passed one at a time

Functions with Multiple Arguments

- Consider that we extend the calculus with the add primitive operation
- The λ-term(lambda(x)(lambda(y)((add x) y))) can be used to add two arguments E_1 and E_2:
- ((lambda(x)(lambda(y)((add x) y))) e1 e2)

beta reduces to

((add e1) e2)   ;; oddly enough, this is claimed to be The Answer…

- The arguments are passed one at a time to make multiple arguments "Currying"..

Functions with Multiple Arguments, some of which are MISSING

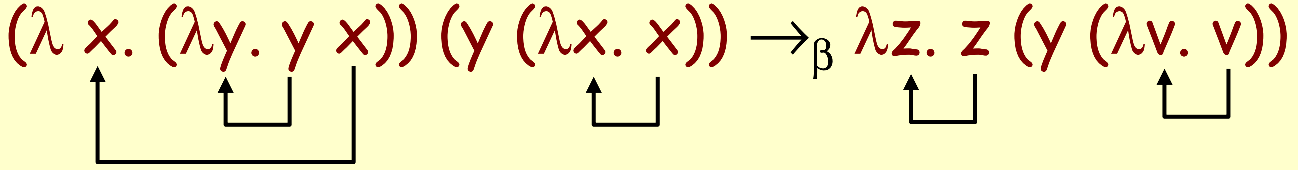
- What is the result of $(\lambda x.\ \lambda y.\ \text{add } x\ y)\ E$?
  - It is $\lambda y.$ add E y

    (A function that given a value E' for y will compute add E E')
- The function $\lambda x.\ \lambda y.\ E$ when applied to one argument E' computes the function $\lambda y.\ [E'/x]E$
- This is one example of <u>higher-order</u> computation
  - We write a function whose result is another function
  - In lisp notation, (add 3) would be a function that takes one argument, say x and adds 3 to it. This is really old stuff from CS61a.

Evaluation and Static Scope

- The definition of substitution guarantees that evaluation respects static scoping:

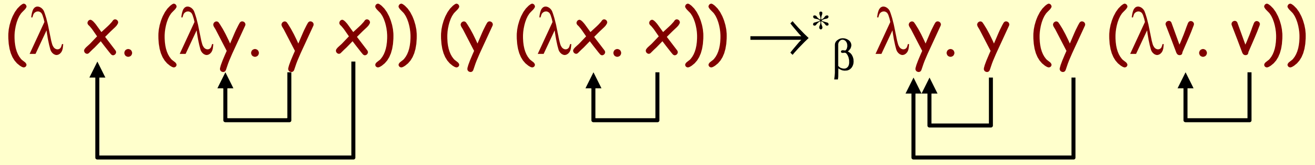
$$(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \to_\beta \lambda z.\ z\ (y\ (\lambda v.\ v))$$



(y remains free, i.e., defined externally)

- If we forget to rename the bound y:

$$(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \to^*_\beta \lambda y.\ y\ (y\ (\lambda v.\ v))$$



(y was free before but is bound now)

The Order of Evaluation

- In a λ-term there could be more than one instance of (λx. E) E'

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

  - could reduce the inner or the outer λ
  - which one should we pick?

$$\begin{tikzcd}
 & (\lambda y.\,(\lambda x.\,x)\,y)\,E \arrow[dl, "\text{inner}"'] \arrow[dr, "\text{outer}"] & \\
(\lambda y.\,[y/x]\,x)\,E = (\lambda y.\,y)\,E \arrow[dr] & & [E/y]\,(\lambda x.\,x)\,y = (\lambda x.\,x)\,E \arrow[dl] \\
 & E &
\end{tikzcd}$$

Order of Evaluation (Cont.)

- The Church-Rosser theorem says that any order will compute the same result
  - A result is a λ-term that cannot be reduced further
- But we might want to fix the order of evaluation when we model a certain language
- In (typical) programming languages we do not reduce the bodies of functions (under a λ)
  - functions are considered values

Call by Name

- Do not evaluate under a λ
- Do not evaluate the argument prior to call
- Example:

$(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$

$(\lambda x.\ x)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$

$(\lambda u.\ u)\ (\lambda v.\ v) \rightarrow_{\beta n}$

$\lambda v.\ v$

Call by Value

- Do not evaluate under λ
- Evaluate an argument prior to call
- Example:

$$(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \to_{\beta v}$$
$$(\lambda y.\ (\lambda x.\ x)\ y)\ (\lambda v.\ v) \to_{\beta v}$$
$$(\lambda x.\ x)\ (\lambda v.\ v) \to_{\beta v}$$
$$\lambda v.\ v$$

Call by Name and Call by Value

- CBN
  - difficult to implement
  - order of side effects not predictable
- CBV:
  - easy to implement efficiently
  - might not terminate even if CBN might terminate
  - Example: $(\lambda x.\ \lambda z.z)\ ((\lambda y.\ yy)\ (\lambda u.\ uu))$
- Outside the functional programming language community only CBV is used

Lambda Calculus and Programming Languages

- Pure lambda calculus has only functions
- What if we want to compute with booleans, numbers, lists, etc.? Like 3+4=7?
- All these can be encoded in pure λ-calculus
- The trick: do not encode what a value is but what we can do with it!
- For each data type we have to describe how it can be used, as a function
  - then we write that function in λ-calculus

Encoding Booleans in Lambda Calculus

- What can we do with a boolean?
  - we can make a binary choice
- A boolean is a function that given two choices selects one of them
  - true $=_{def}$ $\lambda x.\ \lambda y.\ x$
  - false $=_{def}$ $\lambda x.\ \lambda y.\ y$
  - if E_1 then E_2 else E_3 $=_{def}$ $E_1\ E_2\ E_3$
- Example: if true then u else v is

  $(\lambda x.\ \lambda y.\ x)\ u\ v \rightarrow_\beta (\lambda y.\ u)\ v \rightarrow_\beta u$

Encoding Booleans in Lambda Calculus **Lisp notation**

- What can we do with a boolean?
  - we can make a binary choice
- A boolean is a function that given two choices selects one of them
  - true $=_{def}$ ((lambda(x)(lambda(y) x))
  - false $=_{def}$ ((lambda(x)(lambda(y) y))
  - if E_1 then E_2 else E_3 $=_{def}$ ((E_1 E_2)E_3)
- Example: if true then u else v is

  ((lambda(x)(lambda(y) x)) u) v) $\rightarrow_\beta$ ((lambda(y) u) v) $\rightarrow_\beta$ u

Encoding Pairs in Lambda Calculus

- What can we do with a pair?
  - we can select one of its elements
- A pair is a function that given a boolean returns the left or the right element

  cons x y $=_{def}$ λ b. b x y (lambda(b)((b x) y))

  car p $=_{def}$ p true (p true)

  cdr p $=_{def}$ p false (p false)
- Example:

  (car (cons x y)) $\to$ ((cons x y) true) $\to$ (true x y) $\to$ x

  You may recall having seen this in CS61a!

Encoding Natural Numbers in Lambda Calculus

- What can we do with a natural number?
  - we can iterate a number of times
- A natural number is a function that given an operation f and a starting value s, applies f a number of times to s:

  $0 =_{def} \lambda f.\ \lambda s.\ s$

  $1 =_{def} \lambda f.\ \lambda s.\ f\ s$

  $2 =_{def} \lambda f.\ \lambda s.\ f\ (f\ s)$

  and so on

Encoding Natural Numbers in Lisp

- What can we do with a natural number?
  - we can iterate a number of times
- A natural number is a function that given an operation f and a starting value s, applies f a number of times to s:

  0 $=_{def}$ (lambda(f) (lambda(s) s)

  1 $=_{def}$ (lambda(f) (lambda(s) (f s))

  2 $=_{def}$ (lambda(f) (lambda(s) (f(f s)))

  and so on

Computing with Natural Numbers

- The successor function

  $\text{succ } n =_{def} \lambda f.\ \lambda s.\ f\ (n\ f\ s)$
- Addition

  $\text{add } n_1\ n_2 =_{def} n_1 \text{ succ } n_2$
- Multiplication

  $\text{mult } n_1\ n_2 =_{def} n_1\ (\text{add } n_2)\ 0$
- Testing equality with 0

  $\text{iszero } n =_{def} n\ (\lambda b.\ \text{false})\ \text{true}$

Computing with Natural Numbers in lisp notation

- The successor function

  succ n $=_{def}$
  (lambda(f)(lambda(s) (f ((n f) s)))
- Addition

  add n_1 n_2 $=_{def}$ ((n_1 succ) n_2)
- Multiplication

  mult n_1 n_2 $=_{def}$ ((n_1 (add n_2)) 0)

Computing with Natural Numbers. Example

mult 2 2 $\rightarrow$

2 (add 2) 0 $\rightarrow$

(add 2) ((add 2) 0) $\rightarrow$

2 succ (add 2 0) $\rightarrow$

2 succ (2 succ 0) $\rightarrow$

succ (succ (succ (succ 0))) $\rightarrow$

succ (succ (succ (λf. λs. f (0 f s)))) $\rightarrow$

succ (succ (succ (λf. λs. f s))) $\rightarrow$

succ (succ (λg. λy. g ((λf. λs. f s) g y)))

succ (succ (λg. λy. g (g y))) $\rightarrow^*$ λg. λy. g (g (g (g y))) = 4

Computing with Natural Numbers. Example

- What is the result of the application add 0 ?

  $(\lambda n_1.\ \lambda n_2.\ n_1\ succ\ n_2)\ 0 \ \to_\beta$

  $\lambda n_2.\ 0\ succ\ n_2 =$

  $\lambda n_2.\ (\lambda f.\ \lambda s.\ s)\ succ\ n_2 \to_\beta$

  $\lambda n_2.\ n_2 =$

  $\lambda x.\ x$

- By computing with functions we can express some optimizations

Expressiveness of Lambda Calculus

- The λ-calculus can express
  - data types (integers, booleans, lists, trees, etc.)
  - branching (using booleans)
  - recursion
- This is enough to encode Turing machines
- Encodings are fun
- But programming in pure λ-calculus is painful.
- A more fruitful approach for language theorists is to extend the pure λ calculus
  - actually add constants (0, 1, 2, …, true, false, if-then-else, etc.)
  - types

We'll quit here for lack of time

- Much more in CS 263
- Vast literature