Foundations of Programming Languages:
Introduction to Lambda Calculus

Lecture Outline

- Why study lambda calculus?

- Lambda calculus
  - Syntax
  - Evaluation
  - Relationship to programming languages

- Next time: type systems for lambda calculus

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

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 & Russel (Principia Mathematica) used the notation $\hat{x}$ P to denote the set of x's such that P holds

- Church borrowed the notation but moved ^ down to create ^x E

- Which later turned into λx. E and the calculus became known as lambda calculus

Function Application

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

$$E_1 \; E_2$$

  to denote the application of function E_1 to actual argument E_2
- All functions are applied to a single argument

Why Study Lambda Calculus?

- λ-calculus has had a tremendous influence on the design and analysis of programming languages
- 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

Why Study Lambda Calculus?

- λ-calculus 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

Why Study Lambda Calculus?

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

(Landin 1966)

Syntax of Lambda Calculus

- Only three kinds of expressions

E ::= x	variables
| E_1 E_2	function application
| λx. E	function creation

- The form λx. E is also called lambda abstraction, or simply <u>abstraction</u>
- E are called λ-terms or λ-expressions

Examples of Lambda Expressions

- The identity function:

$$I =_{def} \lambda x.\ x$$

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

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

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

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

Notational Conventions

- Application associates to the left

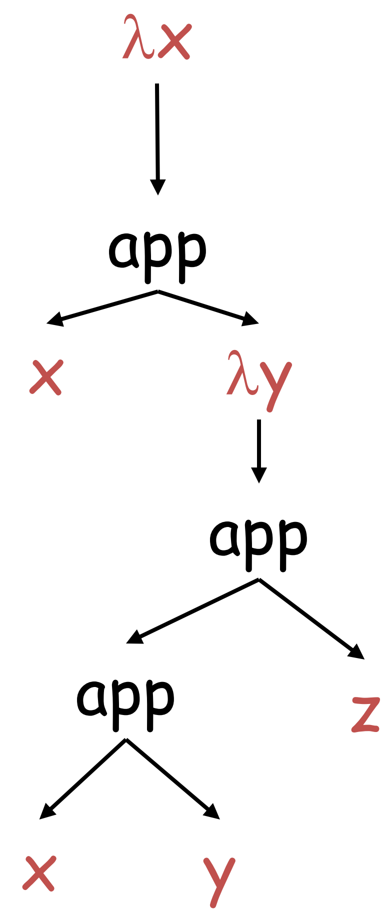
  x y z parses as (x y) z

- 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:



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 $\lambda x.\ E$ <u>binds</u> variable x in E
  - x is the newly introduced variable
  - E is the scope of x
  - we say x is <u>bound</u> in $\lambda x.\ E$
  - Just like formal function arguments are bound in the function body

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:

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

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

$$\text{Free}(\lambda x.\ E) = \text{Free}(E) - \{ x \}$$

- Example: $\text{Free}(\lambda x.\ x\ (\lambda y.\ x\ y\ z)) = \{ z \}$
- Free variables are declared outside the term

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., in Cool: let x ← E in x + (let x ← E’ in x) + x
- In λ-calculus: λx. x (λx. x) x

Renaming Bound Variables

- Two λ-terms that can be obtained from each other by a renaming of the bound variables are considered identical
- Example: λx. x is identical to λy. y and to λz. z
- Intuition:
  - by changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function does not change
  - in λ-calculus such functions are considered identical

Renaming Bound Variables (Cont.)

- Convention: we will always rename bound variables so that they are all unique
  - e.g., write λ x. x (λ y.y) x instead of λ x. x (λ x.x) x
- This makes it easy to see the scope of bindings
- And also prevents serious confusion !

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))

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:

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

- Another example with the identity:

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

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

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

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

- A non-terminating evaluation:

  $(\lambda x.\ xx)(\lambda x.\ xx) \to$

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

Functions with Multiple Arguments

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

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

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

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

  $[E_2/y]\ \mathrm{add}\ E_1\ y\ = \mathrm{add}\ E_1\ E_2$

- The arguments are passed one at a time

Functions with Multiple Arguments

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

    (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 higher-order computation
  - We write a function whose result is another function

Evaluation and the Static Scope

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

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

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

- If we forget to rename the bound y:

$$(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \rightarrow^{*}_{\beta} \lambda y.\ y\ (y\ (\lambda v.\ v))$$

(y was free before but is bound now)

The Order of Evaluation

- In a λ-term, there could be more than one instance of $(\lambda\ x.\ E)\ E'$

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

  – could reduce the inner or the outer \lambda

  – which one should we pick?

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

inner / outer

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

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

E

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:

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

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

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

$\lambda v.\ v$

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.?
- 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 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

  mkpair x y $=_{def}$ λ b. x y

  fst p $=_{def}$ p true

  snd p $=_{def}$ p false

- Example:

  fst (mkpair x y) → (mkpair x y) true → true x y → x

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

Computing with Natural Numbers

- The successor function

  succ n $=_{def}$ λf. λs. 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

- Testing equality with 0

  iszero n $=_{def}$ n (λb. false) true

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 ?

  $(\lambda n_1.\ \lambda n_2.\ n_1\ \mathrm{succ}\ n_2)\ 0 \ \rightarrow_\beta$

  $\lambda n_2.\ 0\ \mathrm{succ}\ n_2 =$

  $\lambda n_2.\ (\lambda f.\ \lambda s.\ s)\ \mathrm{succ}\ n_2 \rightarrow_\beta$

  $\lambda n_2.\ n_2 =$

  $\lambda x.\ x$

- 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
  - we will add constants (0, 1, 2, …, true, false, if-then-else, etc.)
  - and we will add types