CS 611: Lecture 9

More Lambda Calculus: Recursion, Scope, and Substitution

September 17, 1999

Cornell University Computer Science Department
Andrew Myers

Last time

- Introduced compact, powerful programming language: untyped lambda calculus (Church, 1930's)
- All values are *first-class, anonymous functions*
- Syntax: $e ::= \underset{\text{var}}{x} \mid \underset{\text{application}}{e_0\ e_1} \mid \underset{\text{abstraction}}{\lambda\, x\, e_0}$
- Missing:
  - multiple arguments ✓
  - local variables ✓
  - primitive values (booleans, integers, …) ✓
  - data structures ✓
  - recursive functions (this lecture)
  - assignment

Lambda Calculus References

- Gunter (recommended text)
- Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*
- Davie, *An Introduction to Functional Programming Systems using Haskell*
- Barendregt, *The Lamba Calculus: Its Syntax and Semantics*

Recursion

- How to express recursive functions?
- Consider factorial function:

$$\text{fact}(x) = \begin{cases} 1 & \text{if } x = 0 \\ x*\text{fact}(x-1) & \text{if } x > 0 \end{cases}$$

- Can't write this recursive definition!

FACT ↦
(λx (*IF* (*ZERO?* x) 1 (* x (*FACT* (- x 1)))))

(*ZERO?* ↦ (λ n ((n (λ x *FALSE*)) *TRUE*)))

Recursive definitions

FACT ≜
(λx (*IF* (*ZERO?* x) 1 (* x (*FACT* (- x 1)))))

- This is an *equation*, not a definition!
- Meaning:
  FACT stands for a function that, if applied to an argument, gives the same result that
  (λx (*IF* (*ZERO?* x) 1 (* x (*FACT* (- x 1)))))
  does.

Defining a recursive function

- Idea: introduce a function *FACT′* just like *FACT* except that it has an extra argument that should be passed a function f such that ((f f) x) computes factorial of x

FACT′ ≜
(λf (λx (*IF* (*ZERO?* x) 1 (* x ((f f) (- x 1)))))))

- Now define *FACT* ≡ (*FACT′ FACT′*)
- This expression diverges but its application to a number does not!

Evaluation of FACT

FACT′ ≜
(λf (λx (*IF* (*ZERO?* x) 1 (* x ((f f) (- x 1))))))
FACT ≜ (*FACT′ FACT′*)

(*FACT* 2) → ((*FACT′ FACT′*) 2)
→ (*IF* (*ZERO?* 2) 1 (* 2 ((*FACT′ FACT′*) (- 2 1))))
→ (* 2 ((*FACT′ FACT′*) (- 2 1))))))
= (* 2 (*FACT* (- 2 1))))))

Generalizing

FACT′ ≜
(λf (λx (*IF* (*ZERO?* x) 1 (* x ((f f) (- x 1))))))

- The recursion-removal transformation:
  - Add an extra argument variable (f) to the recursive function
  - Replace all internal references to the recursive function with an application of argument variable to itself
  - Replace all external references to the recursive function as application of it to itself
- Can this transformation itself be abstracted?

Fixed point operator

- Suppose we had an operator *Y* that found the fixed point of functions:

$$((Y\ f)\ x) = (f\ ((Y\ f)\ x))$$
$$(Y\ f) = f\ (Y\ f)$$

- Now write a recursive function as a function that takes itself as an argument:

FACTEQN ≡ λf (λx (*IF* (*ZERO?* x) 1 (* x (f (- x 1))))))
Idea: *FACT* = (*FACTEQN FACT*)

FACT ≡ (*Y FACTEQN*)
= *FACTEQN*(*Y FACTEQN*) =
FACTEQN(*FACTEQN*(*FACTEQN*(*FACTEQN*(...))))

Y ?

- Can we express the Y operator as a lambda expression? Maybe not!
- # functions from Z to boolean: $2^{|Z|}$
- # functions: $\geq 2^{|Z|}$
- Set of all functions is *uncountably* infinite
- Set of *computable* functions is the same size: *countably* infinite
- Only an infinitesimal fraction of all functions are computable
- No reason to expect an arbitrary function to be computable! (*e.g.*, halting function is not)

Definition of Y

- Y is a solution to this equation:

$$Y = (\lambda f\ (f\ (Y\ f)))$$

- Now, apply our recursion-removal trick:

$$Y' \equiv (\lambda y\ (\lambda f\ (f\ (\ (y\ \ y)\ f))))$$
$$Y \equiv (Y'\ Y')$$

- Traditional form for *Y*:

$$Y \equiv (\lambda f\ ((\lambda x\ (f\ \ (x\ x))$$
$$(\lambda x\ (f\ \ (x\ x)))$$

Variable binding

- Which variable is denoted by an identifier?
- Lexical scope:

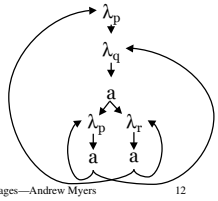
(λ x (λ x x) x)

(λ p (λ q ((λ p (p q)) (λ r (p r)))))

(λ p (λ q ((λ p (p q)) (λ r (p r)))))

(λ • (λ • ((λ • (• •)) (λ • (• •)))))

Stoy diagram



Problems with substitution

- Rule for evaluating an application:

$$((\lambda\, x\, e_1)\, e_2) \rightarrow e_1[e_2 / x]$$

- Can't just stick e_2 in for every occurrence of variable x :

$$(x\ (\lambda\ x\ x))\ [\ (b\ a) / x\ ]\ = ((b\ a)\ (\lambda\ x\ (b\ a)))$$

- Can't just stick e_2 in for every occurrence of variable x outside any lambda over x :

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

Variable capture

A recurring problem!

- Nobody gets the substitution rule right: Church, Hilbert, Gödel, Quine, Newton, etc.
- Substitution problem also comes up most PL's, even in *integral* calculus:

e.g. how to substitute $y := x$ in $y + \int xy\, dx$

- Need to distinguish between *free* and *bound* identifiers—variable capture occurs when we substitute an expression into a context where its variables are bound

Free identifiers

- The function $FV[\![e]\!]$ gives the set of all free (unbound) identifiers in e
- Special brackets $[\![\ ]\!]$ are called *semantic brackets*
  - distinguish functions that operate on the result of expression e and functions that operate on abstract syntax tree for e
  - traditional

$$FV[\![x]\!] = \{ x \}$$
$$FV[\![e_0\ e_1]\!] = FV[\![e_0]\!] \cup FV[\![e_1]\!]$$
$$FV[\![\lambda\, x\, e]\!] = FV[\![e]\!] - \{ x \}$$

Defining substitution inductively

$$x[e / x] \equiv e$$
$$y[e / x] \equiv y \qquad (y \neq x)$$
$$(e_0\ e_1)\ [e_2 / x\ ] \equiv (e_0[e_2/x\ ]\ \ e_1[e_2/x\ ])$$
$$(\lambda\, x\, e_0)[e_1 / x\ ] \equiv (\lambda\, x\, e_0)$$

Substitution into abstraction

$$(\lambda\, y\, e_0)\ [e_1 / x\ ] \triangleq (\lambda\, y'\, e_0[y'/y][e_1/x\ ])$$

where

$$y' \notin FV[\![e_0]\!]$$
$$y' \notin FV[\![e_1]\!]$$
$$y' \neq x$$

Renaming

- Intuitively, meaning of lambda expression does not depend on name of argument variable: $(\lambda\, x\, x) = (\lambda\, y\, y) = (\lambda \bullet \bullet)$

$$(\lambda\, x\, (\lambda\, y\, (y\ x)) = (\lambda\, p\, (\lambda\, q\, (q\ p)) = (\lambda\, y\, (\lambda\, x\, (x\ y))$$
$$= (\lambda \bullet (\lambda \bullet (\bullet\ \bullet)))$$

$$(\lambda\, y\, e_0)\ [e_1 / x\ ] \mapsto (\lambda\, y'\, e_0[y'/y][e_1/x\ ])$$

α-*reduction*: $(\lambda\, y\, e_0) \xrightarrow{\alpha} (\lambda\, y'\, e_0[y'/y])$
where $y' \notin FV[\![e_0]\!]$
(does not change Stoy diagram)

Extensionality

- Two functions are equal by extension if they give the same result when applied to the same argument
- Therefore, the expressions $(\lambda\, x\, (e\, x))$ and e are equal by extension

$$((\lambda\, x\, (e_0\, x))\, e_1) = e_0\, e_1 \quad \text{if } (x \notin FV[\![e_0]\!])$$

- η-reduction: $(\lambda\, x\, (e\, x)) \xrightarrow{\eta} e$ if $(x \notin FV[\![e]\!])$

Reductions

- Three reductions that preserve the meaning of a lambda calculus expression:

$$(\lambda\, x\, e) \xrightarrow{\alpha} (\lambda\, x'\, e[x'/x]) \quad \text{if } (x' \notin FV[\![e]\!])$$
$$((\lambda\, x\, e_1)\, e_2) \xrightarrow{\beta} e_1[e_2\, /\, x]$$
$$(\lambda\, x\, (e\, x)) \xrightarrow{\eta} e \quad \text{if } (x \notin FV[\![e]\!])$$

- Reductions can be applied to sub-expressions: if X reduces to Y then
  - (X Z) reduces to (Y Z) for any Z
  - (Z X) reduces to (Z Y) for any Z
  - $(\lambda\, z\, X)$ reduces to $(\lambda\, z\, Y)$ for any identifier z

Equivalence

- Two lambda expressions are α-equivalent if they can be converted to each other using α-reductions / have the same Stoy diagrams

$(\lambda\, p\, (\lambda\, q\, (q\, p)) \xrightarrow{\alpha} (\lambda\, x\, (\lambda\, q\, (q\, x)) \xrightarrow{\alpha} (\lambda\, x\, (\lambda\, y\, (y\, x))$

- Lambda expressions form *equivalence classes* defined by their Stoy diagrams

Normal Form

- A lambda expression is in *normal form* when no reductions can be performed on it or on any of its sub-expressions
- Reducible expressions are called *redexes*
- All normal forms for an expression are in the same equivalence class!
- What is the normal form for Y ?