CS 611
Advanced Programming Languages

Andrew Myers
Cornell University

Lecture 8: Recursion, scope, and substitution

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

Notation

- Lambda calculus is programming language *and* a mathematical notation for writing down functions
- When programming language: fully parenthesized
- Identity function program: $(\lambda\, x\, x)$
- Mathematical identity function: $f(x) = x$ operating on elements of set T is

$$f \;=\; \lambda x \in T.\, x$$

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)))))

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 as does
  (λx (*IF* (*ZERO?* x) 1 (* x (*FACT* (- x 1)))))

Defining a recursive function

- Idea: introduce a function 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′*)
- *FACT* 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′*)
~ (λf (λx (*IF* (*ZERO?* x) 1 (* x ((*FACT* (- x 1))))))

(*FACT* 3) = ((*FACT′ FACT′*) 3)

→* (*IF* (*ZERO?* 3) 1 (* 3 ((*FACT′ FACT′*) (- 3 1))))
→* (* 3 ((*FACT′ FACT′*) (- 3 1))))))
→* 6

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*(...))))

Is Y computable?

- 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 as Z: *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 = (λf (f (*Y* f)))

- Now, apply our recursion-removal trick:

Y′ ≜ (λy (λf (f ((y y) f))))
Y ≜ (*Y′ Y′*)

- Traditional form for *Y* (requires call-by-name):

Y ≡ (λf ((λx (f (x x))
(λx (f (x x)))

Problems with substitution

- Rule for evaluating an application:

  $((\lambda\, x\, e_1)\; e_2) \to e_1\{e_2 \,/\, x\}$
- Can't just stick e_2 in for every occurrence of variable x :

  (x (λ x x)) { (b a) / x } = ((b a) (λ x (b a)))
- Can't just stick e_2 in for every occurrence of variable x outside any lambda over x :

  (y (λ x (x y))) { x / y } = (x (λ 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 for x in $xy + \int xy\, dx$

- 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 variables

- The function $FV[\![e]\!]$ gives the set of all free variables (unbound identifiers) in e
- Special brackets $[\![\;]\!]$ are called *semantic brackets*; wrap syntactic arguments
  - $FV[\![e]\!]$ operates on abstract syntax tree for e, not result of evaluating e
  - sometimes name of function is omitted: $[\![e]\!]$
- Inductive definition of $FV[\![\cdot]\!]$:

$FV[\![x]\!] = \{\, x \,\}$

$FV[\![e_0\; e_1]\!] = FV[\![e_0]\!] \cup FV[\![e_1]\!]$

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

Defining substitution inductively

Let $e\{e'/x\} \to e''$ mean "e'' can be the result of substituting e' for x"

$x\{e \,/\, x\} \;\to\; e$

$y\{e \,/\, x\} \;\to\; y \quad (\text{if } y \neq x)$

$(e_0\; e_1)\,\{e_2/x\} \;\to\; (e_0\{e_2/x\}\;\; e_1\{e_2/x\}\,)$

$(\lambda\, x\, e_0)\{e_1/x\} \;\to\; (\lambda\, x\, e_0)$

Substitution into abstraction

$$(\lambda y\;\; e_0)\;\{e_1/x\} \to (\lambda y\;\; e_0\{e_1/x\})$$

$$(\text{if } y \neq x,\; y \notin FV[\![e_1]\!])$$

$$(\lambda y\;\; e_0)\;\{e_1/x\} \to (\lambda y'\; e_0\{y'/y\}\{e_1/x\})$$

$$(\text{if } y' \notin FV[\![e_1]\!],\; y' \notin FV[\![e_0]\!],\; y' \neq x)$$

Variable binding

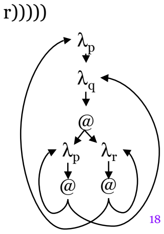
- Which variable is denoted by an identifier? (λ x (λ x x) x)
- Lexical scope:

  (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 • (• •)))))

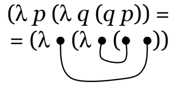
Stoy diagram



Renaming

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

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



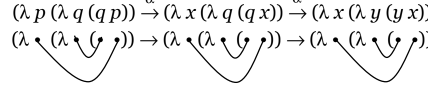
$$(\lambda\, y\, e_0)\, \{e_1 / x\} = (\lambda\, y'\, e_0\{y'/y\}\{e_1/x\})$$

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)

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