Programming Languages and Compilers (CS 421)

William Mansky

http://courses.engr.illinois.edu/cs421/

Based in part on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, Elsa Gunter, and Dennis Griffith

Lambda Calculus

- Aims to capture the essence of functions, function applications, and evaluation
- λ-calculus is a theory of computation
- Programs may be viewed as functions from input (initial state and input values) to output (resulting state and output values)
- λ-calculus makes this precise, and provides rules for working with functions in general
- Can be typed or untyped, but we'll focus on untyped

Lambda Calculus: Motivation

- Typed and untyped λ-calculus used for theoretical study of (sequential) programming languages
- Programming languages can be thought of as λ-calculus + predefined constructs, constants, types, syntactic sugar (denotational semantics)
- OCaml is close to the λ-calculus:

$$\text{fun x -> exp} \equiv \lambda\, x.\ \text{exp}$$

$$\text{let } x = e_1 \text{ in } e_2 \equiv (\lambda\, x.\ e_2)\ e_1$$

Untyped λ-calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e

    (Function creation, as fun x -> e)
  - Application: e_1 e_2

Untyped λ-calculus Grammar

- Formal BNF Grammar:
  - <expression> ::= <variable>
    | <abstraction>
    | <application>
    | (<expression>)
  - <abstraction>
    ::= λ<variable>.<expression>
  - <application>
    ::= <expression> <expression>

Example

- Which variables are which?

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

Example

- Which variables are which?

$$\begin{tikzcd}[column sep=tiny]
(\lambda & x. & y & \lambda & y. & y \arrow[l, bend right] & (\lambda & x. & x \arrow[l, bend right] & y) \arrow[lllll, bend right] & x) \arrow[lllllllll, bend left] & x
\end{tikzcd}$$

Working with Variables

- Occurrence: a location of a subterm (variable or complex term) in a term
- Variable binding: λ x. e is a binding of x to e
- Bound occurrence: all occurrences of x in λ x. e
- Free occurrence: one that is not bound
- Scope of binding: in λ x. e, all occurrences in e not in a subterm of the form λ x. e' (same x) are in scope of that binding of x
- Free variables: all variables with free occurrences in a term

Untyped λ-calculus

- How do you compute with the λ-calculus?
- Roughly speaking, by substitution:

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

- Modulo subtleties to avoid free variable capture

Semantics of Substitution

- $x\ [e / x] = e$
- $y\ [e / x] = y$ if $y \neq x$

- $(e_1\ e_2)\ [e / x] = ((e_1\ [e / x]\ )\ (e_2\ [e / x]\ ))$

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

if $y \neq x$ and y is not a free variable in e

Changing Names

- α-conversion:

$$\lambda\ x.\ e \overset{\alpha}{\rightarrow} \lambda\ y.\ (e\ [y/x])$$

- Provided that

1. y is not free in e
2. No free occurrence of x in e becomes bound when replaced by y

α-conversion Failure

1. Error: y is free in term

$$\lambda\, x.\, x\, y \overset{\alpha}{\to} \lambda\, y.\, y\, y$$

2. Error: free occurrence of x becomes bound in wrong way when replaced by y

$$\lambda\, x.\, \lambda\, y.\, x \overset{\alpha}{\to} \lambda\, y.\, \lambda\, y.\, y$$

But $\lambda\, x.\, (\lambda\, y.\, y)\, x \overset{\alpha}{\to} \lambda\, y.\, (\lambda\, y.\, y)\, y$

and $\lambda\, y.\, (\lambda\, y.\, y)\, y \overset{\alpha}{\to} \lambda\, x.\, (\lambda\, y.\, y)\, x$ are okay

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if
- ~ is an equivalence relation (reflexive, symmetric, transitive)
- If $e_1 \sim e_2$ then
  - $(e\ e_1) \sim (e\ e_2)$ and $(e_1\ e) \sim (e_2\ e)$
  - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$
- Congruent terms are "functionally the same" in some way

α-equivalence

- α-equivalence is the smallest congruence containing α-conversion
  - i.e., two terms are α-equivalent if they can be α-converted into the same term
- We usually treat α-equivalent terms as *equal*

Proving α-equivalence

Show: λ x. (λ y. y x) x $=_\alpha$ λ y. (λ x. x y) y

- λ x. (λ y. y x) x $\overset{\alpha}{\rightarrow}$ λ z. (λ y. y z) z, so

  λ x. (λ y. y x) x $=_\alpha$ λ z. (λ y. y z) z

- (λ y. y z) $\overset{\alpha}{\rightarrow}$ (λ x. x z), so

  (λ y. y z) $=_\alpha$ (λ x. x z) and

  λ z. (λ y. y z) z $=_\alpha$ λ z. (λ x. x z) z

- λ z. (λ x. x z) z $\overset{\alpha}{\rightarrow}$ λ y. (λ x. x y) y, so

  λ z. (λ x. x z) z $=_\alpha$ λ y. (λ x. x y) y

Semantics of Substitution

- $x\ [e / x] = e$
- $y\ [e / x] = y$ if $y \neq x$

- $(e_1\ e_2)\ [e / x] = ((e_1\ [e / x]\ )\ (e_2\ [e / x]\ ))$

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

if $y \neq x$ and y is not a free variable in e

α-convert here if necessary!

Substitution Example

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

Substitution Example

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

- Problems?
  - y free in the residue

Substitution Example

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

- Problems?
  - y free in the residue
- $(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z]$

$\overset{\alpha}{\rightarrow} (\lambda\ w.\ w\ z)\ [(\lambda\ x.\ x\ y)\ /\ z]$

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

Substitution Example

- Only replace free occurrences
- $(\lambda\ y.\ y\ z\ (\lambda\ z.\ z))\ [(\lambda\ x.\ x)\ /\ z] = ?$

Substitution Example

- Only replace free occurrences
- $(\lambda\ y.\ y\ z\ (\lambda\ z.\ z))\ [(\lambda\ x.\ x)\ /\ z] =$

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

Not

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

β reduction

- β Rule: $(\lambda\ x.\ P)\ N \overset{\beta}{\rightarrow} P\ [N\ /x]$

- Essence of computation in the lambda calculus

Example

- (λ z. (λ x. x y) z) (λ y. y z)

  $\xrightarrow{\beta}$ (λ x. x y) (λ y. y z)

  $\xrightarrow{\beta}$ (λ y. y z) y $\xrightarrow{\beta}$ y z

- (λ x. x x) (λ x. x x)

  $\xrightarrow{\beta}$ (λ x. x x) (λ x. x x)

  $\xrightarrow{\beta}$ (λ x. x x) (λ x. x x) $\xrightarrow{\beta}$ …

αβ-equivalence

- αβ-equivalence is the smallest congruence containing α-equivalence and β-reduction
- A term is in *normal form* if no subterm is α-equivalent to a term that can be β-reduced
- Theorem (Church-Rosser): if e_1 and e_2 are αβ-equivalent and both are normal forms, then they are α-equivalent
  - So each term has a unique fully reduced form, up to α-equivalence

Order of Evaluation

- Order of evaluation matters!
  - Not all terms reduce to normal forms
  - Not all reduction strategies will produce a normal form if one exists
- Two main strategies: eager and lazy
- Reflected in functional languages (OCaml is eager, Haskell is lazy)

Lazy Evaluation

- Reduce the left side of an application first
- β-reduce when left side is an abstraction (function)
- Don't evaluate the right side unless we have to!
- When there are multiple applications, go top-down and left-to-right

Lazy Example

- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y)) \xrightarrow{\beta}\ ?$

Lazy Example

- (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))

$\overset{\beta}{\rightarrow}$ (λ x. x)

Done!

Eager Evaluation

- Reduce the left side of an application first
- Then reduce the right side
- β-reduce when left side is an abstraction (function) and right side cannot be reduced (eagerly) any further
  - Might not be a normal form!
- When there are multiple applications, go top-down and left-to-right
- Evaluate everything we can

Eager Example

- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y)) \overset{\beta}{\rightarrow}$?

Eager Example

- (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))

$\overset{\beta}{\rightarrow}$ (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))

Eager Example

- (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))

$\xrightarrow{\beta}$ (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))

$\xrightarrow{\beta}$ (λ z. (λ x. x)) ((λ y. y y) (λ y. y y))

$\xrightarrow{\beta}$ …

Operational Semantics for λ-calculus

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}$$

- Application (version 1 - Lazy Evaluation)

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

- Application (version 2 - Eager Evaluation)

$$\frac{E' \rightarrow E''}{(\lambda\ x\ .\ E\ )\ E' \rightarrow (\lambda\ x\ .\ E\ )\ E''}$$

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

where V is a variable or abstraction

η (Eta) Reduction

- η Rule: λ x. e x $\overset{\eta}{\rightarrow}$ e if x not free in e
  - Can be useful in both directions
  - Not valid in OCaml
    - Recall lambda-lifting and side effects
  - Different from (λ x. e) x $\rightarrow$ e (β-reduction)

- Example: λ x. (λ y. y) x $\overset{\eta}{\rightarrow}$ λ y. y

Expressiveness

- Untyped λ-calculus is Turing Complete
  - Can express any sequential computation
- Tricky parts:
  - How to express basic data: booleans, integers, etc?
  - How to express recursion?
  - Constants, if_then_else, etc, are conveniences; can be added as syntactic sugar

Typed vs. Untyped λ-calculus

- The *pure* λ-calculus has no notion of type: (f f) is a legal expression
- Types restrict which applications are valid
- Types are not syntactic sugar! They disallow some terms/executions
- Simply typed λ-calculus is less powerful than the untyped λ-calculus: NOT Turing Complete (no recursion)