

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

[http://www.cs.uiuc.edu/class
/fa05/cs421/current/](http://www.cs.uiuc.edu/class/fa05/cs421/current/)

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

Where Are We?

- First four weeks
 - Functional Programming
 - Intro to OCAML
 - Recursion
 - Higher-Order Functions
 - User defined datatypes

Where Are We?

- Types and Type Systems
 - Unification
 - Type Derivation and Inference
- Language Syntax and Parsing
 - DFAs and NDFAS, Lexing
 - Grammars
 - First and Follow Sets
 - LL Grammars
 - LR Grammars

Where Are We Going?

- Next three weeks
 - Lambda Calculus
 - Evaluation
 - Modeling Programs in Lambda Calculus (Denotational Semantics)
- Later in course
 - Natural Semantics
 - Transition Semantics
- Interpreters - Implementing Programming Language Semantics
- And some more ...

Lambda Calculus - Motivation

- Aim is to capture the essence of functions, function applications, and evaluation

$\forall \lambda$ -calculus is a theory of computation

- “The Lambda Calculus: Its Syntax and Semantics”. H. P. Barendregt. North Holland, 1984

Lambda Calculus - Motivation

- All *sequential programs* may be viewed as functions from input (initial state and input values) to output (resulting state and output values).

\forall λ -calculus is a mathematical formalism of functions and functional computations

- Two flavors: typed and untyped

Untyped λ -Calculus

- Only three kinds of expressions:
 - Variables: x, y, z, w, \dots
 - Abstraction: $\lambda x. e$
(Function creation, think `fun x -> e`)
 - Application: $e_1 e_2$

Untyped λ -Calculus Grammar

- Formal BNF Grammar:
 - $\langle \text{expression} \rangle \rightarrow \langle \text{variable} \rangle$
 - | $\langle \text{abstraction} \rangle$
 - | $\langle \text{application} \rangle$
 - | $(\langle \text{expression} \rangle)$
 - $\langle \text{abstraction} \rangle \rightarrow \lambda \langle \text{variable} \rangle . \langle \text{expression} \rangle$
 - $\langle \text{application} \rangle \rightarrow \langle \text{expression} \rangle \langle \text{expression} \rangle$

Untyped λ -Calculus

Terminology

- **Occurrence**: a location of a subterm in a term
- **Variable binding**: $\lambda x. e$ is a binding of x in e
- **Bound occurrence**: all occurrences of x in $\lambda x. e$
- **Free occurrence**: one that is not bound
- **Scope of binding**: in $\lambda x. e$, all occurrences in e not in a subterm of the form $\lambda x. e'$ (same x)
- **Free variables**: all variables having free occurrences in a term

Example

- Label occurrences and scope:

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

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 all kinds of subtleties to avoid free variable capture

How Powerful is the Untyped λ -Calculus?

- The untyped λ -calculus is Turing Complete
 - Can express any sequential computation
- Problems:
 - 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
- Simply typed λ -calculus is less powerful than the untyped λ -Calculus: NOT Turing Complete (no recursion)

Uses of λ -Calculus

- Typed and untyped λ -calculus used for theoretical study of sequential programming languages
- Sequential programming languages are essentially the λ -calculus, extended with predefined constructs, constants, types, and syntactic sugar
- Ocaml is close to the λ -Calculus:
 $\text{fun } x \rightarrow \text{exp} \quad \dashrightarrow \quad \lambda x. \text{exp}$
 $\text{let } x = e_1 \text{ in } e_2 \quad \dashrightarrow \quad (\lambda x. e_2)e_1$

α Conversion

- α -conversion:
$$\lambda x. \text{exp} \xrightarrow{\alpha} \lambda y. (\text{exp} [y/x])$$
- Provided that
 1. y is not free in exp
 2. No free occurrence of x in exp becomes bound in exp when replaced by y

α Conversion Non-Examples

1. y is not free in exp

$$\lambda x. x y \not\rightarrow_{\alpha} \lambda y. y y$$

2. No free occurrence of x becomes bound when replaced by x

$$\lambda x. \underbrace{\lambda y. x y}_{\text{exp}} \not\rightarrow_{\alpha} \lambda y. \underbrace{\lambda y. y y}_{\text{exp}[y/x]}$$

But $\lambda x. (\lambda y. y) x \rightarrow_{\alpha} \lambda y. (\lambda y. y) y$

And $\lambda y. (\lambda y. y) y \rightarrow_{\alpha} \lambda x. (\lambda y. y) x$

Congruence

- Let \sim be a relation on lambda terms. \sim is a **congruence** if
- it is an equivalence relation
- If $e_1 \sim e_2$ then
 - $(e e_1) \sim (e e_2)$ and $(e_1 e) \sim (e_2 e)$
 - $\lambda x. e_1 \sim \lambda x. e_2$

α Equivalence

- α equivalence is the smallest congruence containing α conversion
- One usually treats α -equivalent terms as equal - i.e. use α equivalence classes of terms

Example

Show: $\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda y. (\lambda x. x y) y$

$\forall \lambda x. (\lambda y. y x) x \rightarrow_{\alpha} \lambda z. (\lambda y. y z) z$ so

$\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda z. (\lambda y. y z) z$

• $(\lambda y. y z) \rightarrow_{\alpha} (\lambda x. x z)$ so

$\lambda z. (\lambda y. y z) z \sim_{\alpha} \lambda z. (\lambda x. x z) z$

$\forall \lambda z. (\lambda x. x z) z \rightarrow_{\alpha} \lambda y. (\lambda x. x y) y$ so

$\lambda z. (\lambda x. x z) z \sim_{\alpha} \lambda y. (\lambda x. x y) y$

$\forall \lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda y. (\lambda x. x y) y$

η (Eta) Reduction

- η Rule: $\lambda x. f x \xrightarrow{\eta} f$ if x not free in f
 - Can be useful in each direction
 - Not valid in Ocaml
 - recall lambda-lifting and side effects
 - Not equivalent to $(\lambda x. f) x \rightarrow f$ (inst of β)
- Example: $\lambda x. (\lambda y. y) x \xrightarrow{\eta} \lambda y. y$

Substitution

- Defined on α -equivalence classes of terms
- $[N / x] P$ means replace every free occurrence of x in P by N
- Provided that no variable free in P becomes bound in $[N / x] P$
 - Rename bound variables in P to avoid capturing free variables of N

Substitution

- $[N / x] x = N$
- $[N / x] y = y$ if $y \neq x$
- $[N / x] (e_1 e_2) = (([N / x] e_1) ([N / x] e_2))$
- $[N / x] (\lambda x. e) = (\lambda x. e)$
- $[N / x] (\lambda y. e) = \lambda y. ([N / x] e)$
provided $y \neq x$ and y not free in N
–Rename y if necessary

Example

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

- Problems?

- z in redex in scope of y binding

- y free in the residue

- $[(\lambda x. x y) / z] (\lambda y. y z) \xrightarrow{\alpha} \lambda$

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

Example

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

Not

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

β reduction

- β Rule: $(\lambda x. P) N \xrightarrow{\beta} [N / x] P$
- Essence of computation in the lambda calculus
- Usually defined on α -equivalence classes of terms

Example

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

$--\beta--> (\lambda x. x y) (\lambda y. y z)$

$--\beta--> (\lambda y. y z) y --\beta--> y z$

• $(\lambda x. x x) (\lambda x. x x)$

$--\beta--> (\lambda x. x x) (\lambda x. x x)$

$--\beta--> (\lambda x. x x) (\lambda x. x x) --\beta--> \dots$

α β Equivalence

- \forall α β 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
 - Hard fact (Church-Rosser): if e_1 and e_2 are $\alpha\beta$ -equivalent and both are normal forms, then they are α equivalent

Order of Evaluation

- Not all terms reduce to normal forms
- Not all reduction strategies will produce a normal form if one exists

Lazy evaluation:

- Always reduce the left-most application in a top-most series of applications (i.e. Do not perform reduction inside an abstraction)
- Stop when left-most application is not an application of an abstraction to a term

Example 1

- $(\lambda z. (\lambda x. x)) ((\lambda y. y y) (\lambda y. y y))$
- Lazy evaluation:
- Reduce the left-most application:
- $(\lambda z. (\lambda x. x)) ((\lambda y. y y) (\lambda y. y y))$
 $\rightarrow_{\beta} (\lambda x. x)$

Eager evaluation

- (Eagerly) reduce left of top application to an abstraction
- Then (eagerly) reduce argument
- Then β -reduce the application

Example 1

- $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
- Eager evaluation:
- Reduce the rator of the top-most application to an abstraction: Done.
- Reduce the argument:
- $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
- β --> $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
- β --> $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))\dots$

Example 2

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

- Lazy evaluation:

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

Example 2

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

- Lazy evaluation:

$(\lambda x. \boxed{x} \boxed{x}) \underline{((\lambda y. y y) (\lambda z. z))} \xrightarrow{\beta}$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

$\boxed{((\lambda y. y y) (\lambda z. z))} \boxed{((\lambda y. y y) (\lambda z. z))}$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

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

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

$((\lambda y. \boxed{y} \boxed{y}) \underline{(\lambda z. z)}) ((\lambda y. y y) (\lambda z. z))$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

$((\lambda y. \boxed{y} \boxed{y}) \underline{(\lambda z. z)}) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}\rightarrow (\boxed{(\lambda z. z)} \boxed{(\lambda z. z)})((\lambda y. y y) (\lambda z. z))$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

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

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

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

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

$\text{--}\beta\text{--}\rightarrow ((\lambda z. \boxed{z}) (\underline{\lambda z. z}))((\lambda y. y y) (\lambda z. z))$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

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

$\text{--}\beta\text{--}\rightarrow ((\lambda z. \boxed{z}) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}\rightarrow \boxed{(\lambda z. z)} ((\lambda y. y y) (\lambda z. z))$

Example 2

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

- Lazy evaluation:

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

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

$\text{--}\beta\text{--}\rightarrow ((\lambda z. z) (\lambda z. z))((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}\rightarrow (\lambda z. \boxed{z}) ((\lambda y. y y) (\lambda z. z)) \text{--}\beta\text{--}\rightarrow$

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

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

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

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

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

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

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

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

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

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

$(\lambda y. y y) (\lambda z. z) \sim_{\beta} \lambda z. z$

Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Eager evaluation:

