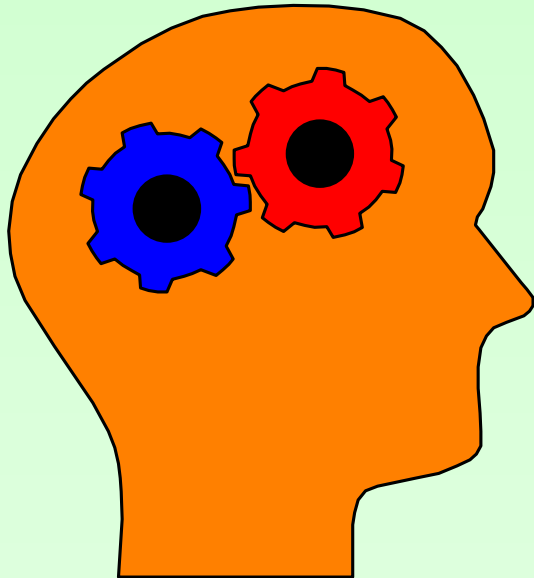




CS6202: Advanced Topics in Programming Languages and Systems

Lecture 1 : **Lambda Calculus**



Lambda Calculus

- Untyped Lambda Calculus
- Evaluation Strategy
- Techniques - encoding, extensions, recursion
- Operational Semantics
- Explicit Typing
- Type Rules and Type Assumption
- Progress, Preservation, Erasure

Introduction to Lambda Calculus:

<http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf>

<http://www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/geuvers.pdf>

Untyped Lambda Calculus

- Extremely simple programming language which captures *core* aspects of computation and yet allows programs to be treated as mathematical objects.
- Focused on *functions* and applications.
- Invented by Alonzo (1936,1941), used in programming (Lisp) by John McCarthy (1959).

Functions without Names

Usually functions are given a name (e.g. in language C):

```
int plusone(int x) { return x+1; }  
...plusone(5)...
```

However, function names can also be dropped:

```
(int (int x) { return x+1; } ) (5)
```

Notation used in untyped lambda calculus:

```
( $\lambda$  x. x+1) (5)
```

Syntax

In purest form (no constraints, no built-in operations), the lambda calculus has the following syntax.

$t ::=$		terms
x		variable
$\lambda x . t$		abstraction
$t t$		application

This is **simplest** universal programming language!

Conventions

- Parentheses are used to avoid ambiguities.
e.g. $x y z$ can be either $(x y) z$ or $x (y z)$
- Two conventions for avoiding too many parentheses:
 - Applications associates to the left
e.g. $x y z$ stands for $(x y) z$
 - Bodies of lambdas extend as far as possible.
e.g. $\lambda x. \lambda y. x y x$ stands for $\lambda x. (\lambda y. ((x y) x))$.
- Nested lambdas may be collapsed together.
e.g. $\lambda x. \lambda y. x y x$ can be written as $\lambda x y. x y x$

Scope

- An occurrence of variable x is said to be *bound* when it occurs in the body t of an abstraction $\lambda x . t$
- An occurrence of x is *free* if it appears in a position where it is not bound by an enclosing abstraction of x .

- Examples: $x y$

$\lambda y . x y$

$\lambda x . x$

(identity function)

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

(non-stop loop)

$(\lambda x . x) y$

$(\lambda x . x) x$

Alpha Renaming

- Lambda expressions are equivalent up to bound variable renaming.

$$\begin{aligned} \text{e.g. } \lambda x. x &=_{\alpha} \lambda y. y \\ \lambda y. x y &=_{\alpha} \lambda z. x z \end{aligned}$$

But NOT:

$$\lambda y. x y \neq_{\alpha} \lambda y. z y$$

- Alpha renaming rule:

$$\lambda x. E =_{\alpha} \lambda z. [x \mapsto z] E \quad (z \text{ is not free in } E)$$

Beta Reduction

- An application whose LHS is an abstraction, evaluates to the body of the abstraction with parameter substitution.

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

- Beta reduction rule (operational semantics):

$$(\lambda x. t_1) t_2 \rightarrow_{\beta} [x \mapsto t_2] t_1$$

Expression of form $(\lambda x. t_1) t_2$ is called a *redex* (reducible expression).

Evaluation Strategies

- A term may have many redexes. Evaluation strategies can be used to limit the number of ways in which a term can be reduced.
- An evaluation strategy is *deterministic*, if it allows reduction with at most one redex, for any term.
- Examples:
 - full beta reduction
 - normal order
 - call by name
 - call by value, etc

Full Beta Reduction

- Any redex can be chosen, and evaluation proceeds until no more redexes found.
- Example: $(\lambda x.x) ((\lambda x.x) (\lambda z. (\lambda x.x) z))$
denoted by $\text{id} (\text{id} (\lambda z. \text{id} z))$

Three possible redexes to choose:

$\text{id} (\text{id} (\lambda z. \text{id} z))$

$\text{id} (\text{id} (\lambda z. \text{id} z))$

$\text{id} (\text{id} (\lambda z. \text{id} z))$

- Reduction:

$\text{id} (\text{id} (\lambda z. \text{id} z))$

$\rightarrow \text{id} (\text{id} (\lambda z.z))$

$\rightarrow \text{id} (\lambda z.z)$

$\rightarrow \lambda z.z$

\nrightarrow

Normal Order Reduction

- Deterministic strategy which chooses the *leftmost, outermost* redex, until no more redexes.
- Example Reduction:

$$\begin{aligned} & \underline{\text{id (id (\lambda z. id z))}} \\ & \rightarrow \underline{\text{id (\lambda z. id z)}} \\ & \rightarrow \lambda z. \underline{\text{id z}} \\ & \rightarrow \lambda z. z \\ & \not\rightarrow \end{aligned}$$

Call by Name Reduction

- Chooses the *leftmost, outermost* redex, but *never* reduces inside abstractions.
- Example:

$$\begin{aligned} & \underline{\text{id (id (\lambda z. id z))}} \\ & \rightarrow \underline{\text{id (\lambda z. id z)}} \\ & \rightarrow \lambda z. \text{id z} \\ & \nrightarrow \end{aligned}$$

Call by Value Reduction

- Chooses the *leftmost, innermost* redex whose RHS is a value; and never reduces inside abstractions.
- Example:

$$\begin{aligned} & \text{id } (\underline{\text{id } (\lambda z. \text{id } z)}) \\ & \rightarrow \underline{\text{id } (\lambda z. \text{id } z)} \\ & \rightarrow \lambda z. \text{id } z \\ & \nrightarrow \end{aligned}$$

Strict vs Non-Strict Languages

- *Strict* languages always evaluate all arguments to function before entering call. They employ call-by-value evaluation (e.g. C, Java, ML).
- *Non-strict* languages will enter function call and only evaluate the arguments as they are required. *Call-by-name* (e.g. Algol-60) and *call-by-need* (e.g. Haskell) are possible evaluation strategies, with the latter avoiding the re-evaluation of arguments.
- In the case of call-by-name, the evaluation of argument occurs with each parameter access.

Programming Techniques in λ -Calculus

- Multiple arguments.
- Church Booleans.
- Pairs.
- Church Numerals.
- Enrich Calculus.
- Recursion.

Multiple Arguments

- Pass multiple arguments one by one using lambda abstraction as intermediate results. The process is also known as *currying*.
- Example:

$$f = \lambda(x,y).s \quad \longrightarrow \quad f = \lambda x. (\lambda y. s)$$

Application:

$f(v,w)$

*requires pairs as
primitive types*

$(f v) w$

*requires higher
order feature*

Church Booleans

- Church's encodings for true/false type with a conditional:

true = $\lambda t. \lambda f. t$

false = $\lambda t. \lambda f. f$

if = $\lambda l. \lambda m. \lambda n. l m n$

- Example:

if true v w

= $(\lambda l. \lambda m. \lambda n. l m n) \text{ true } v w$

→ true v w

= $(\lambda t. \lambda f. t) v w$

→ v

- Boolean and operation can be defined as:

and = $\lambda a. \lambda b. \text{if } a \text{ b false}$

= $\lambda a. \lambda b. (\lambda l. \lambda m. \lambda n. l m n) a b \text{ false}$

= $\lambda a. \lambda b. a b \text{ false}$

Pairs

- Define the functions `pair` to construct a pair of values, `fst` to get the first component and `snd` to get the second component of a given pair as follows:

`pair` = $\lambda f. \lambda s. \lambda b. b f s$

`fst` = $\lambda p. p \text{ true}$

`snd` = $\lambda p. p \text{ false}$

- Example:

`snd (pair c d)`

= $(\lambda p. p \text{ false}) ((\lambda f. \lambda s. \lambda b. b f s) c d)$

→ $(\lambda p. p \text{ false}) (\lambda b. b c d)$

→ $(\lambda b. b c d) \text{ false}$

→ `false c d`

→ `d`

Church Numerals

- Numbers can be encoded by:

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s z$$

$$c_2 = \lambda s. \lambda z. s (s z)$$

$$c_3 = \lambda s. \lambda z. s (s (s z))$$

:

Church Numerals

- Successor function can be defined as:

$$\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)$$

Example:

$$\begin{aligned} & \text{succ } c_1 \\ &= (\lambda n. \lambda s. \lambda z. s (n s z)) (\lambda s. \lambda z. s z) \\ &\rightarrow \lambda s. \lambda z. s ((\lambda s. \lambda z. s z) s z) \\ &\rightarrow \lambda s. \lambda z. s (s z) \end{aligned}$$

$$\begin{aligned} & \text{succ } c_2 \\ &= \lambda n. \lambda s. \lambda z. s (n s z) (\lambda s. \lambda z. s (s z)) \\ &\rightarrow \lambda s. \lambda z. s ((\lambda s. \lambda z. s (s z)) s z) \\ &\rightarrow \lambda s. \lambda z. s (s (s z)) \end{aligned}$$

Church Numerals

- Other Arithmetic Operations:

plus = $\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$

times = $\lambda m. \lambda n. m (\text{plus } n) c_0$

iszero = $\lambda m. m (\lambda x. \text{false}) \text{true}$

- Exercise : Try out the following.

plus c_1 x

times c_0 x

times x c_1

iszero c_0

iszero c_2

Enriching the Calculus

- We can add constants and built-in primitives to enrich λ -calculus. For example, we can add boolean and arithmetic constants and primitives (e.g. true, false, if, zero, succ, iszero, pred) into an enriched language we call λNB :
- Example:
 - $\lambda x. \text{succ} (\text{succ } x) \in \lambda\text{NB}$
 - $\lambda x. \text{true} \in \lambda\text{NB}$

Recursion

- Some terms go into a loop and do not have normal form.

Example:

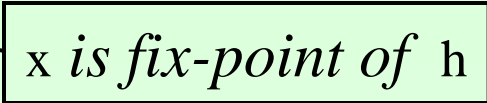
$$\begin{aligned} & (\lambda x. x x) (\lambda x. x x) \\ \rightarrow & (\lambda x. x x) (\lambda x. x x) \\ \rightarrow & \dots \end{aligned}$$

- However, others have an interesting property

$$\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$$

which returns a fix-point for a given functional.

Given $x = h x$
 $= \text{fix } h$



That is: $\text{fix } h \rightarrow h (\text{fix } h) \rightarrow h (h (\text{fix } h)) \rightarrow \dots$


Example - Factorial

- We can define factorial as:

$\text{fact} = \lambda n. \text{if } (n \leq 1) \text{ then } 1 \text{ else times } n \text{ (fact (pred } n))$

$= (\lambda h. \lambda n. \text{if } (n \leq 1) \text{ then } 1 \text{ else times } n \text{ (h (pred } n))) \text{ fact}$

$= \text{fix } (\lambda h. \lambda n. \text{if } (n \leq 1) \text{ then } 1 \text{ else times } n \text{ (h (pred } n)))$



Example - Factorial

- Recall:
fact = fix ($\lambda h. \lambda n. \text{if } (n \leq 1) \text{ then } 1 \text{ else times } n \text{ (h (pred n))}$)
- Let $g = (\lambda h. \lambda n. \text{if } (n \leq 1) \text{ then } 1 \text{ else times } n \text{ (h (pred n))}$)

Example reduction:

$$\begin{aligned} \text{fact } 3 &= \text{fix } g \ 3 \\ &= g \ (\text{fix } g) \ 3 \\ &= \text{times } 3 \ ((\text{fix } g) \ (\text{pred } 3)) \\ &= \text{times } 3 \ (g \ (\text{fix } g) \ 2) \\ &= \text{times } 3 \ (\text{times } 2 \ ((\text{fix } g) \ (\text{pred } 2))) \\ &= \text{times } 3 \ (\text{times } 2 \ (g \ (\text{fix } g) \ 1)) \\ &= \text{times } 3 \ (\text{times } 2 \ 1) \\ &= 6 \end{aligned}$$

Formal Treatment of Lambda Calculus

- Let V be a countable set of variable names. The set of terms is the smallest set T such that:
 1. $x \in T$ for every $x \in V$
 2. if $t_1 \in T$ and $x \in V$, then $\lambda x. t_1 \in T$
 3. if $t_1 \in T$ and $t_2 \in T$, then $t_1 t_2 \in T$

- Recall syntax of lambda calculus:

$t ::=$	terms
x	variable
$\lambda x.t$	abstraction
$t t$	application

Free Variables

- The set of free variables of a term t is defined as:

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

$$\text{FV}(\lambda x.t) = \text{FV}(t) \setminus \{x\}$$

$$\text{FV}(t_1 t_2) = \text{FV}(t_1) \cup \text{FV}(t_2)$$

Substitution

- Works when free variables are replaced by term that does not clash:

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

- However, problem if there is name capture/clash:

$$[x \mapsto \lambda z. z w] (\lambda x. x) \neq (\lambda x. \lambda z. z w)$$

$$[x \mapsto \lambda z. z w] (\lambda w. x) \neq (\lambda w. \lambda z. z w)$$

Formal Defn of Substitution

$$[x \mapsto s] x = s \quad \text{if } y=x$$

$$[x \mapsto s] y = y \quad \text{if } y \neq x$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$[x \mapsto s] (\lambda y.t) = \lambda y.t \quad \text{if } y=x$$

$$[x \mapsto s] (\lambda y.t) = \lambda y. [x \mapsto s] t \quad \text{if } y \neq x \wedge y \notin \text{FV}(s)$$

$$[x \mapsto s] (\lambda y.t) = [x \mapsto s] (\lambda z. [y \mapsto z] t) \\ \text{if } y \neq x \wedge y \in \text{FV}(s) \wedge \text{fresh } z$$

Syntax of Lambda Calculus

- Term:

$t ::=$

x

$\lambda x.t$

$t t$

terms

variable

abstraction

application

- Value:

$t ::=$

$\lambda x.t$

terms

abstraction value

Call-by-Value Semantics

$$\begin{array}{c} \text{premise} \swarrow \\ \frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \end{array} \quad (\text{E-App1})$$

conclusion \nearrow

$$\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \quad (\text{E-App2})$$

$$(\lambda x.t) v \rightarrow [x \mapsto v] t \quad (\text{E-AppAbs})$$

Getting Stuck

- Evaluation can get stuck. (Note that only values are λ -abstraction)

e.g. $(x\ y)$

- In extended lambda calculus, evaluation can also get stuck due to the absence of certain primitive rules.

$(\lambda x. \text{succ } x)\ \text{true} \rightarrow \text{succ true} \not\rightarrow$

Boolean-Enriched Lambda Calculus

- Term:

$t ::=$	terms
x	variable
$\lambda x.t$	abstraction
$t t$	application
true	constant true
false	constant false
if t then t else t	conditional

- Value:

$v ::=$	value
$\lambda x.t$	abstraction value
true	true value
false	false value

Key Ideas

- Exact typing impossible.

if <long and tricky expr> then true else ($\lambda x.x$)

- Need to introduce function type, but need argument and result types.

if true then ($\lambda x.true$) else ($\lambda x.x$)

Simple Types

- The set of simple types over the type Bool is generated by the following grammar:
- $T ::=$
 - Bool type of booleans
 - $T \rightarrow T$ type of functions
- \rightarrow is right-associative:

$T_1 \rightarrow T_2 \rightarrow T_3$ denotes $T_1 \rightarrow (T_2 \rightarrow T_3)$

Implicit or Explicit Typing

- Languages in which the programmer declares all types are called *explicitly typed*. Languages where a typechecker infers (almost) all types is called *implicitly typed*.
- Explicitly-typed languages places onus on programmer but are usually better documented. Also, compile-time analysis is simplified.

Explicitly Typed Lambda Calculus

- $t ::=$ terms
 - ...
 - $\lambda x : T.t$ abstraction
 - ...
- $v ::=$ value
 - $\lambda x : T.t$ abstraction value
 - ...
- $T ::=$ types
 - Bool type of booleans
 - $T \rightarrow T$ type of functions

Examples

true

$\lambda x:\text{Bool} . x$

$(\lambda x:\text{Bool} . x) \text{ true}$

if false then $(\lambda x:\text{Bool} . \text{True})$ else $(\lambda x:\text{Bool} . x)$

Erasure

- The erasure of a simply typed term t is defined as:

$$\begin{aligned}\text{erase}(x) &= x \\ \text{erase}(\lambda x : \mathbf{T}.t) &= \lambda x. \text{erase}(t) \\ \text{erase}(t_1 t_2) &= \text{erase}(t_1) \text{erase}(t_2)\end{aligned}$$

- A term m in the untyped lambda calculus is said to be *typable* in λ_{\rightarrow} (simply typed λ -calculus) if there are some simply typed term t , type T and context Γ such that:

$$\text{erase}(t)=m \wedge \Gamma \vdash t : T$$

Typing Rule for Functions

- First attempt:

$$\frac{t_2 : T_2}{\lambda x:T_1. t_2 : T_1 \rightarrow T_2}$$

- But $t_2:T_2$ can assume that x has type T_1

Need for Type Assumptions

- Typing relation becomes ternary

$$\frac{x:T_1 \vdash t_2 : T_2}{\lambda x:T_1.t_2 : T_1 \rightarrow T_2}$$

- For nested functions, we may need several assumptions.

Typing Context

- A *typing context* is a finite map from *variables to their types*.
- Examples:

$x : \text{Bool}$

$x : \text{Bool}, y : \text{Bool} \rightarrow \text{Bool}, z : (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool}$

Type Rule for Abstraction

Shall use Γ to denote typing context.

$$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-Abs})$$

Other Type Rules

- Variable

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-Var})$$

- Application

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2} \quad (\text{T-App})$$

- Boolean Terms.

Typing Rules

True : Bool (T-true)

False : Bool (T-false)

0 : Nat (T-Zero)

$$\frac{t_1:\text{Bool} \quad t_2:T \quad t_3:T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-If})$$

$$\frac{t : \text{Nat}}{\text{succ } t : \text{Nat}} \quad (\text{T-Succ})$$

$$\frac{t : \text{Nat}}{\text{pred } t : \text{Nat}} \quad (\text{T-Pred})$$

$$\frac{t : \text{Nat}}{\text{iszero } t : \text{Bool}} \quad (\text{T-Iszero})$$

Example of Typing Derivation

$$x : \text{Bool} \in x : \text{Bool}$$
$$\frac{}{x : \text{Bool} \vdash x : \text{Bool}} \quad (\text{T-Var})$$
$$\frac{}{\vdash (\lambda x : \text{Bool}. x) : \text{Bool} \rightarrow \text{Bool}} \quad (\text{T-Abs})$$
$$\frac{}{\vdash \text{true} : \text{Bool}} \quad (\text{T-True})$$
$$\frac{}{\vdash (\lambda x : \text{Bool}. x) \text{ true} : \text{Bool}} \quad (\text{T-App})$$

Canonical Forms

- If v is a value of type `Bool`, then v is either `true` or `false`.
- If v is a value of type $T_1 \rightarrow T_2$, then $v = \lambda x:T_1. t_2$ where $t:T_2$

Progress

Suppose t is a closed well-typed term (that is $\{\} \vdash t : T$ for some T).

Then either t is a value or else there is some t' such that $t \rightarrow t'$.

Preservation of Types (under Substitution)

If $\Gamma, x:S \vdash t : T$ and $\Gamma \vdash s : S$

then $\Gamma \vdash [x \mapsto s]t : T$

Preservation of Types (under reduction)

If $\Gamma \vdash t : T$ and $t \rightarrow t'$

then $\Gamma \vdash t' : T$

Motivation for Typing

- Evaluation of a term either results in a *value* or *gets stuck!*
- Typing can *prove* that an expression cannot get stuck.
- Typing is *static* and can be checked at compile-time.

Normal Form

A term t is a *normal form* if there is no t' such that $t \rightarrow t'$.

The multi-step evaluation relation \rightarrow^* is the reflexive, transitive closure of one-step relation.

pred (succ(pred 0))	
\rightarrow	
pred (succ 0)	pred (succ(pred 0))
\rightarrow	\rightarrow^*
0	0

Stuckness

Evaluation may fail to reach a value:

succ (if true then false else true)

→

succ (false)

↯

A term is *stuck* if it is a normal form but not a value.

Stuckness is a way to characterize *runtime errors*.

Safety = Progress + Preservation

- Progress : A well-typed term is not stuck. Either it is a value, or it can take a step according to the evaluation rules.

Suppose t is a well-typed term (that is $t:T$ for some T).
Then either t is a value or else there is some t' with $t \rightarrow t'$

Safety = Progress + Preservation

- Preservation : If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

If $t:T \wedge t \rightarrow t'$ then $t':T$.