CS2104

Lambda Calculus :

A Simplest Universal Programming Language

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. λ x. λ y. x y x stands for λ x. (λ y. ((x y) x)).

- Nested lambdas may be collapsed together.
  e.g. λ x. λ y. x y x can be written as λ 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 λ 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	
λy. x y	
λ x. x	(identity function)
(λ x. x x) (λ x. x x)	(non-stop loop)
(λ x. x) y	
(λ x. x) x	

Alpha Renaming

- Lambda expressions are equivalent up to bound variable renaming.

  e.g. $\lambda\, x.\, x \quad =_\alpha \quad \lambda\, y.\, y$

  $\lambda\, y.\, x\, y \quad =_\alpha \quad \lambda\, z.\, x\, z$

  But NOT:

  $\lambda\, y.\, x\, y \quad =_\alpha \quad \lambda\, y.\, z\, y$

- Alpha renaming rule:

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

Beta Reduction

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

  e.g.

  $(\lambda\ x.\ x\ y)\ z \quad \to_\beta \quad z\ y$

  $(\lambda\ x.\ y)\ z \quad \to_\beta \quad y$

  $(\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x) \quad \to_\beta \quad (\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$

- Beta reduction rule (operational semantics):

$$(\ \lambda\ x\ .\ t_1\ )\ t_2 \quad \to_\beta \quad [x \mapsto t_2]\ t_1$$

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:
  - normal order
  - call by name
  - call by value, etc

Normal Order Reduction

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

- Example Reduction:

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

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

$\to \lambda z.\underline{\text{id z}}$

$\to \lambda z.z$

$\not\to$

Call by Name Reduction

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

- Example:

$\underline{\text{id (id } (\lambda z.\ \text{id } z))}$
$\to \underline{\text{id } (\lambda z.\ \text{id } z))}$
$\to \lambda z.\text{id } z$
$\not\to$

Call by Value Reduction

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

- Example:

$$\text{id } (\underline{\text{id } (\lambda z.\ \text{id } z)})$$
$$\to \underline{\text{id } (\lambda z.\ \text{id } z)}$$
$$\to \lambda z.\text{id } z$$
$$\not\to$$

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.

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:

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

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

$$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$$

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

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

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

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

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

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

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

Syntax of Lambda Calculus

- Term:

t ::=		terms
	x	variable
	λ x.t	abstraction
	t t	application

- Value:

t ::=		terms
	λ x.t	abstraction value

Oz Abstract Syntax Tree

Distfix notation

t ::=	terms
x	variable
λ x . t	abstraction
t t	application

Oz notation

<T> ::=	terms
x	variable
lam(x <T>)	abstraction
app(<T> <T>)	application
let(x#<T> <T>)	let binding

Why Oz AST?

- Need to program in Oz!

- Unambiguous

x y z → (x y) z → app(app(x y) z)

x y z → x (y z) → app(x app(y z))

Call-by-Value Semantics

premise

$$\frac{t_1 \to t'_1}{t_1\ t_2 \to t'_1\ t_2} \qquad \text{(E-App1)}$$

conclusion

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

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

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.

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

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:

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

Application:

f(v,w) | (f v) w

requires pairs as primitve types | *requires higher order feature*

Church Booleans

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

  true = λ t. λ f. t

  false = λ t. λ f. f

  if = λ l. λ m. λ n. l m n

- Example:

  if true v w

  = (λ l. λ m. λ n. l m n) true v w

  $\rightarrow$ true v w

  = (λ t. λ f. t) v w

  $\rightarrow$ v

- Boolean and operation can be defined as:

  and = λ a. λ b. if a b false

  = λ a. λ b. (λ l. λ m. λ n. l m n) a b false

  = λ a. λ b. a b 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:

$$\begin{aligned} \text{pair} &= \lambda\, f.\ \lambda\, s.\ \lambda\, b.\ b\ f\ s \\ \text{fst} &= \lambda\, p.\ p\ \text{true} \\ \text{snd} &= \lambda\, p.\ p\ \text{false} \end{aligned}$$

  - Example:

$$\begin{aligned} & \text{snd (pair c d)} \\ =\ \ & (\lambda\, p.\ p\ \text{false})\ ((\lambda\, f.\ \lambda\, s.\ \lambda\, b.\ b\ f\ s)\ c\ d) \\ \rightarrow\ \ & (\lambda\, p.\ p\ \text{false})\ (\lambda\, b.\ b\ c\ d) \\ \rightarrow\ \ & (\lambda\, b.\ b\ c\ d)\ \text{false} \\ \rightarrow\ \ & \text{false}\ c\ d \\ \rightarrow\ \ & d \end{aligned}$$

Church Numerals

- Numbers can be encoded by:

$$
\begin{aligned}
c_0 &= \lambda\, s.\ \lambda\, z.\ z \\
c_1 &= \lambda\, s.\ \lambda\, z.\ s\ z \\
c_2 &= \lambda\, s.\ \lambda\, z.\ s\ (s\ z) \\
c_3 &= \lambda\, s.\ \lambda\, z.\ s\ (s\ (s\ z)) \\
&\vdots
\end{aligned}
$$

Church Numerals

- Successor function can be defined as:

  succ = λ n. λ s. λ z. s (n s z)

  Example:

  succ c_1

  = (λ n. λ s. λ z. s (n s z)) (λ s. λ z. s z)

  $\rightarrow$ λ s. λ z. s ((λ s. λ z. s z) s z)

  $\rightarrow$ λ s. λ z. s (s z)

  succ c_2

  = λ n. λ s. λ z. s (n s z) (λ s. λ z. s (s z))

  $\rightarrow$ λ s. λ z. s ((λ s. λ z. s (s z)) s z)

  $\rightarrow$ λ s. λ z. s (s (s z))

Church Numerals

- Other Arithmetic Operations:

$$\begin{aligned} \text{plus} &= \lambda\ m.\ \lambda\ n.\ \lambda\ s.\ \lambda\ z.\ m\ s\ (n\ s\ z) \\ \text{times} &= \lambda\ m.\ \lambda\ n.\ m\ (\text{plus}\ n)\ c_0 \\ \text{iszero} &= \lambda\ m.\ m\ (\lambda\ x.\ \text{false})\ \text{true} \end{aligned}$$

- 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:

  λ x. succ (succ x) $\in \lambda$NB

  λ x. true $\in \lambda$NB

Recursion

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

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

  $\rightarrow\quad (\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$

  $\rightarrow\quad \ldots$

- 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 $\quad x = h\ x$

  $\quad = \text{fix}\ h$

  x *is fix-point of* h

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

Example - Factorial

- We can define factorial as:

fact = λ n. if (n<=1) then 1 else times n (fact (pred n))

= (λ h. λ n. if (n<=1) then 1 else times n (h (pred n))) fact

= fix (λ h. λ n. if (n<=1) then 1 else times n (h (pred n)))

Example - Factorial

- Recall:

  fact = fix (λ h. λ n. if (n<=1) then 1 else times n (h (pred n)))

- Let g = (λ h. λ n. if (n<=1) then 1 else times n (h (pred n)))

  Example reduction:

  fact 3 = fix g 3
  = g (fix g) 3
  = times 3 ((fix g) (pred 3))
  = times 3 (g (fix g) 2)
  = times 3 (times 2 ((fix g) (pred 2)))
  = times 3 (times 2 (g (fix g) 1))
  = times 3 (times 2 1)
  = 6

Alternative using Let Binding

- Enriched lambda calculus with explicit recursion

let(x#exp1 exp2) $\Longrightarrow$ local x in
  x=exp1
  exp2
end

scope of x is both exp1 and exp2

Example : let (fact # λ n. n. if (n<=1) then 1 else times n (fact (pred n))
in (fact 5)

Boolean-Enriched Lambda Calculus

- Term:

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

- Value:

v ::=		value
	λ x.t	abstraction value
	true	true value
	false	false value

Key Ideas

- Exact typing impossible.

  if <long and tricky expr> then true else (λ x.x)

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

  if true then (λ x.true) else (λ x.x)

Simple Types

- The set of simple types over the type Bool is generated by the following grammar:

- T ::= types

  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
	…	
	λ x : T.t	abstraction
	…	
• v ::=		value
	λ x : T.t	abstraction value
	…	
• T ::=		types
	Bool	type of booleans
	$T \rightarrow T$	type of functions

Examples

true

λ x:Bool . x

(λ x:Bool . x) true

if false then (λ x:Bool . True) else (λ x:Bool . x)

Erasure

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

erase(x)	=	x
erase(λx :T.t)	=	λ x. erase(t)
erase(t_1 t_2)	=	erase(t_1) erase(t_2)

- A term m in the untyped lambda calculus is said to be *typable* in $\lambda_\rightarrow$ (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:

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

- 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

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

- Boolean Terms.

Typing Rules

True : Bool (T-true)

False : Bool (T-false)

0 : Nat (T-Zero)

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

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

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

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

Example of Typing Derivation

$$
\dfrac{\dfrac{\dfrac{x : \mathrm{Bool} \in x : \mathrm{Bool}}{x : \mathrm{Bool} \vdash x : \mathrm{Bool}}\ (\text{T-Var})}{\vdash (\lambda\, x : \mathrm{Bool}.\ x) : \mathrm{Bool} \to \mathrm{Bool}}\ (\text{T-Abs}) \qquad \dfrac{}{\vdash \mathrm{true} : \mathrm{Bool}}\ (\text{T-True})}{\vdash (\lambda\, x : \mathrm{Bool}.\ x)\ \mathrm{true}\ : \mathrm{Bool}}\ (\text{T-App})
$$

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)
$\rightarrow$
0

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

Stuckness

Evaluation may fail to reach a value:

succ (if true then false else true)

$\rightarrow$

succ (false)

$\not\rightarrow$

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 $t'{:}T$.