Lambda Calculus

Lecture (1): Untyped Lambda Calculus

Sriram Rajamani
Microsoft Research

Today...

- Untyped lambda calculus
- Syntax & Operational semantics
- "Programming" with lambda calculus

References: 1. Ben Pierce's book: "Types and programming languages"
2. George Necula's lecture notes

Syntax:

$$
\begin{aligned}
e \;::=\;& x && \leftarrow \text{variable}\\
\mid\;& \lambda x.\, e && \leftarrow \text{function abstraction}\\
\mid\;& e_1 \; e_2 && \leftarrow \text{function application}\\
\mid\;& (e) && \leftarrow \text{bracketed expression}
\end{aligned}
$$

terms

Conventions:

$$e_1 \; e_2 \; e_3 \equiv (e_1 \; e_2) \; e_3 \leftarrow \text{application is left associative}$$

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

scope of λ expands as far right as possible

Examples:

$\lambda x . x$

Identity

$\lambda x . \lambda y . x$

function that takes 2 arguments x & y and returns first argument x

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

h.o. function that takes function f, value x and applies f on x twice

Lambda calculus

We will use lambda calculus to study foundations of type systems

From Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Lambda_calculus

In mathematical logic and computer science, lambda calculus, also λ-calculus, is a formal system designed to investigate function definition, function application, and recursion. It was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s; Church used lambda calculus in 1936 to give a negative answer to the Entscheidungsproblem. Lambda calculus can be used to define what a computable function is. The question of whether two lambda calculus expressions are equivalent cannot be solved by a general algorithm. This was the first question, even before the halting problem, for which undecidability could be proved. Lambda calculus has greatly influenced functional programming languages, such as Lisp, ML and Haskell.

Lambda calculus can be called the smallest universal programming language. It consists of a single transformation rule (variable substitution) and a single function definition scheme. Lambda calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to the Turing machine formalism. However, lambda calculus emphasizes the use of transformation rules, and does not care about the actual machine implementing them. It is an approach more related to software than to hardware.

Entscheidungsproblem

From Wikipedia, the free encyclopedia

http://en.wikipedia.org/wiki/Entscheidungsproblem

In mathematics, the *Entscheidungsproblem* (German for 'decision problem') is a challenge posed by David Hilbert in 1928.

The Entscheidungsproblem asks for a computer program that will take as input a description of a formal language and a mathematical statement in the language and return as output either "True" or "False" according to whether the statement is true or false. The program need not justify its answer, or provide a proof, so long as it is always correct. Such a computer program would be able to decide, for example, whether statements such as the continuum hypothesis or the Riemann hypothesis are true, even though no proof or disproof of these statements is known.

In 1936, Alonzo Church and Alan Turing published independent papers showing that it is impossible to decide algorithmically whether statements in arithmetic are true or false, and thus a general solution to the Entscheidungsproblem is impossible. This result is now known as the Church-Turing Theorem

Scope, binding, bound & free occurrences

term

$$\lambda x.\ \overbrace{x\ y}^{\text{Scope of } x}$$

binder for x

bound occurrence of x

free occurrence of y

Scope, binding, bound & free occurrences

no binder, <u>free</u>

$$\lambda x .\ x \ y$$

binder

Find the bound & free occurrences & binders for bound occurrences

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

Find the bound & free occurrences & binders for bound occurrences

free   free

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

α- renaming

Renaming bound variables

$$\lambda x. x \equiv \lambda y. y \equiv \lambda z. z$$

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

easier to understand, only one binding per variable

de Bruijn notation for λ-terms

de Bruijn index of a variable $\triangleq$ number of λs that separate the occurrence from the binder

de Bruijn notation: replace variable occurrences by de Bruijn indexes

$$\lambda x.\ \lambda y.\ x\ y \equiv \lambda.\ \lambda.\ 1\ 0$$

$$\lambda x.\ \lambda x.\ x \equiv \lambda.\ \lambda.\ 0$$

$$\lambda z.\ \lambda y.\ y \equiv \lambda.\ \lambda.0$$

Combinators

A λ-term without any free variables is a combinator

eg:

$$I = \lambda x.\ x$$

$$K = \lambda x.\ \lambda y.\ x$$

$$S = \lambda f.\ \lambda g.\ \lambda x.\ f\ x\ (g\ x)$$

$$D = \lambda x.\ x\ x$$

$$Y = \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f\ (x\ x))$$

Theorem:

Any combinator is "equivalent" to one written with S, K & I

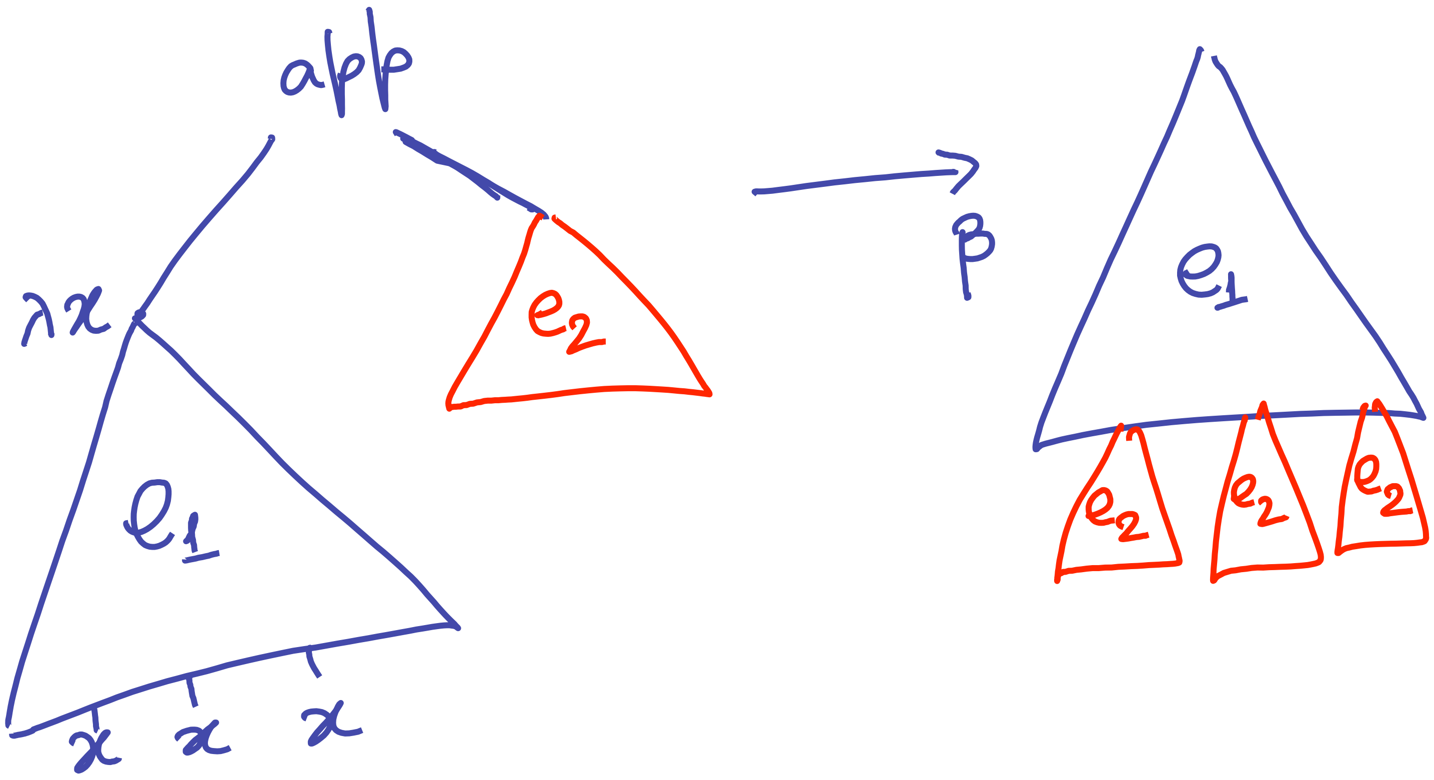
eg: $D =_\beta S\,I\,I$

define later

Operational semantics

$$(\lambda x.\ e_1)\ e_2 \xrightarrow{}_{\beta} \underbrace{[x \mapsto e_2]\ e_1}_{\text{term obtained by replacing all free occ. of } x \text{ in } e_1 \text{ by } e_2}$$

Pictorially . . . . .



Operational semantics

$$(\lambda x.\ e_1)\ e_2 \xrightarrow[\beta]{} \underbrace{[x \mapsto e_2]\ e_1}_{\text{term obtained by replacing all free occ. of } x \text{ in } e_1 \text{ by } e_2}$$

need care in defining substitution

Defining substitution   first attempt

$$[x \mapsto s]\, x = s$$

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

$$[x \mapsto s]\, (\lambda y.e) = \lambda y.\, [x \mapsto s]\, e$$

$$[x \mapsto s]\, (e_1\ e_2) = ([x \mapsto s] e_1)\ \ ([x \to s] e_2)$$

eg1: $[x \mapsto \lambda z.\, z\, w]\, (\lambda y.\, x) =$

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

Defining substitution first attempt

$$[x \mapsto s]\, x = s$$

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

$$[x \mapsto s]\, (\lambda y . e) = \lambda y . [x \mapsto s]\, e$$

$$[x \mapsto s]\, (e_1\ e_2) = ([x \mapsto s] e_1) \ ([x \to s] e_2)$$

eg1: $[x \mapsto \lambda z . z w]\, (\lambda y . x) = \lambda y . \lambda z . z w$ ✓

$[x \mapsto y]\, (\lambda x . x) = \lambda x . y$ ✗

Defining substitution Second attempt

$[x \mapsto s]\, x = s$

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

$$[x \mapsto s]\,(\lambda y . e) = \begin{cases} \lambda y . [x \mapsto s]\, e & \text{if } y \neq x \\ \lambda y . e & \text{if } y = x \end{cases}$$

$[x \mapsto s]\,(e_1\ e_2) = ([x \mapsto s] e_1)\ \ ([x \to s] e_2)$

eg2: $[x \mapsto z]\,(\lambda z . x) =$

Defining substitution   Second attempt

$[x \mapsto s]\, x = s$

$[x \mapsto s]\, y = y$, if $x \neq y$

$$[x \mapsto s]\, (\lambda y. e) = \begin{cases} \lambda y. [x \mapsto s]\, e & \text{if } y \neq x \\ \lambda y. e & \text{if } y = x \end{cases}$$

$[x \mapsto s]\, (e_1\ e_2) = ([x \mapsto s] e_1)\ ([x \to s] e_2)$

eg2: $[x \mapsto z]\, (\lambda z. x) = \lambda z. z$ ✗

capture!

Defining substitution   Second attempt

$[x \mapsto s]\, x = s$

$[x \mapsto s]\, y = y$, if $x \neq y$

$$[x \mapsto s]\,(\lambda y . e) = \begin{cases} \lambda y . [x \mapsto s]\, e & \text{if } y \neq x \wedge y \notin FV(s) \\ \lambda y . e & \text{if } y = x \end{cases}$$

$[x \mapsto s]\,(e_1\ e_2) = ([x \mapsto s] e_1)\ \ ([x \to s] e_2)$

eg2: $[x \mapsto z]\,(\lambda z . x) =$

Defining substitution

final attempt
correct!

$$[x \mapsto s]\, x = s$$

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

$$[x \mapsto s]\, (\lambda y . e) = \begin{cases} \lambda y . [x \mapsto s]\, e & \text{if } y \neq x \wedge y \notin FV(s) \\ \lambda y . e & \text{if } y = x \end{cases}$$

$$[x \mapsto s]\, (e_1\; e_2) = ([x \mapsto s] e_1) \quad ([x \to s] e_2)$$

eg 2: $[x \mapsto z] (\lambda z . x) =$

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

Non-deterministic Operational Semantics

$$\frac{}{(\lambda x.\, e_1)\; e_2 \to_\beta [x \mapsto e_2]\, e_1}$$

$$\frac{e_1 \to_\beta e_1'}{e_1\; e_2 \to_\beta e_1'\; e_2} \qquad \frac{e_2 \to_\beta e_2'}{e_1\; e_2 \to_\beta e_1\; e_2'}$$

$$\frac{e \to_\beta e'}{\lambda x.\, e \to_\beta \lambda x.\, e'}$$

will omit β from now on, but will remember it!

More than one β-reduction sequence possible

$$
\begin{tikzcd}
 & (\lambda y.(\lambda x.x)\,y)\,E \arrow[dl, "\text{inner}"'] \arrow[dr, "\text{outer}"] & \\
(\lambda y.y)\,E \arrow[dr] & & (\lambda x.x)\,E \arrow[dl] \\
 & E &
\end{tikzcd}
$$

A relation $\rightarrow$ has diamond property if

$\forall e_1, e_2, e$ s.t.

$$e \rightarrow e_1$$

$$e \rightarrow e_2$$

$\exists e'$ s.t. $e_1 \rightarrow e'$ and $e_2 \rightarrow e'$

$$\begin{tikzcd}
 & e \arrow[dl] \arrow[dr] & \\
e_1 \arrow[dr] & & e_2 \arrow[dl] \\
 & e' &
\end{tikzcd}$$

such e' should exist

A relation $\to$ has diamond property if

$\forall\, e_1, e_2, e$ s.t.

$e \to e_1$

$e \to e_2$

$\exists\, e'$ s.t. $e_1 \to e'$ and $e_2 \to e'$

$$\begin{tikzcd} & e \arrow[dl] \arrow[dr] & \\ e_1 \arrow[dr] & & e_2 \arrow[dl] \\ & e' & \end{tikzcd}$$

such e' should exist

$\to_\beta$ does not satisfy diamond property

$\to_\beta^*$ satisfies diamond property

Also called CHURCH-ROSSER Thm

Normal form

- A term without β-redexes is in normal form
- β-reduction stops at normal form
- Church-Rosser thm says that independent of reduction strategy we will not find more than one normal form...

... but.. some reduction strategies might fail to find a normal form

$$(\lambda x.y)(\underline{(\lambda y.yy)(\lambda y.yy)})$$

$$\to_\beta (\lambda x.y)(\underline{(\lambda y.yy)(\lambda y.yy)})$$

$$\to_\beta \cdots$$

But ...

$$(\lambda x.y)\underline{((\lambda y.y\,y)(\lambda y.y\,y))}$$

$$\to_\beta (\lambda x.y)\underline{((\lambda y.y\,y)(\lambda y.y\,y))}$$

$$\to_\beta \dots$$

$$\underbrace{(\lambda x.y)((\lambda y.y\,y)(\lambda y.y\,y))}$$

$$\to_\beta y$$

reduction strategies

- Normal order – no reduction under λ
  leftmost outermost redex is reduced.

Thm:
If e has nomal form e', then normal order reduction will reduce e to e'

Call by name

Two rules:

- No reduction inside a λ
- Don't evaluate the argument of a function

$$\frac{}{\lambda x.e \xrightarrow{*}_{n} \lambda x.e} \qquad \frac{e_1 \xrightarrow{*}_{n} \lambda x.e' \quad [x \mapsto e_2]e' \xrightarrow{*}_{n} e}{e_1\ e_2 \xrightarrow{*}_{n} e}$$

Demand driven, expression not evaluated unless it is needed

$$(\lambda y.(\lambda x.x)\ y)((\lambda u.u)(\lambda v.v)) \longrightarrow_{\beta n}$$

$$\underline{(\lambda y.(\lambda x.x)\ y)}((\lambda u.u)(\lambda v.v)) \longrightarrow_{\beta n}$$

$$\underline{(\lambda y.y)((\lambda u.u)(\lambda v.v))} \longrightarrow_{\beta n}$$

$$(\lambda u.u)(\lambda v.v) \longrightarrow_{\beta n}$$

$$(\lambda v.v)$$

Call by value

Two rules:

- No reduction inside a λ
- DO evaluate the argument of a function

$$\frac{}{\lambda x.e \longrightarrow_v^* \lambda x.e} \qquad \frac{e_1 \longrightarrow_v^* \lambda x.e_1' \quad e_2 \longrightarrow_v^* e_2' \quad [e_2' \mapsto x]e_1' \longrightarrow_v^* e}{e_1\ e_2 \longrightarrow_v^* e}$$

Most languages are call by value -

$$(\lambda y.(\lambda x.x)\ y)((\lambda u.u)(\lambda v.v)) \longrightarrow_{\beta n}$$

$$(\lambda y.(\lambda x.x)\ y)\ \underline{((\lambda u.u)\ (\lambda v.v))} \longrightarrow_{\beta_v}$$

$$\underline{(\lambda y.(\lambda x.x)\ y)\ (\lambda v.v)} \longrightarrow_{\beta_v}$$

$$\underline{(\lambda x.x)\ (\lambda v.v)} \longrightarrow_{\beta_v}$$

$$\lambda v.v$$

Programming in the λ-calculus

- λ-calculus is expressive enough to encode turing machines

- Let $=_\beta$ be defined as reflexive, symmetric, transitive closure of $\rightarrow_\beta$

$$\boxed{e \stackrel{?}{=}_\beta e' \text{ is undecidable}}$$

Encoding booleans

$$true =_{def} \lambda x . \lambda y . x$$

$$false =_{def} \lambda x . \lambda y . y$$

$$\text{if } E_1 \text{ then } E_2 \text{ else } E_3 =_{def} E_1\ E_2\ E_3$$

Eg "if true then e_1 else e_2"

$$= ((\lambda x . \lambda y . x)\ \ e_1\ \ e_2) \to_\beta e_1$$

Natural numbers

Church numerals

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

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

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

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

$$\vdots$$

Successor

$$SCC = \lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z)$$

eg $\quad scc\ 3 \stackrel{?}{=}$

Addition

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

$$m \quad s \quad (n \; s \; z)$$

$$plus \;\; 2 \;\; 3 \stackrel{?}{=}$$

Multiplication

$$times = \lambda m.\lambda n.\ \lambda s.\lambda z.\ f_0$$

$$m \quad (plus\ n) \quad C_0$$

Recursion

Fix point combinator:

$$fix = \lambda f. (\lambda x. f (\lambda y. x\ x\ y)) \\ (\lambda x. f (\lambda y. x\ x\ y))$$

$$g = \lambda fct. \lambda n. \text{ if } (eq\ n\ c_0) \text{ then } c_1 \\ \text{else } (times\ n\ (fct\ (pred\ n)))$$

$$factorial = fix\ g$$

Program verification for λ-calculus

- Add types to λ-calculus terms
- Set up a type system that ensures
  "Well-typed terms cannot go wrong"

Next time ...

1. Add bool, nat, succ, pred, if-then-else
   as primitives

2. Add typing ...