



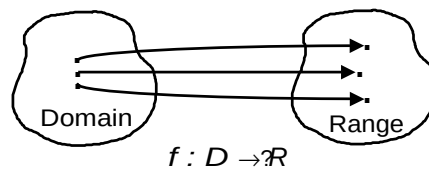
λ -calculus: A Basis for Functional Languages

Arvind
Laboratory for Computer Science
M.I.T.

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Functions



f may be viewed as

- a set of ordered pairs $\langle d, r \rangle$ where $d \in D$ and $r \in R$
- a *method of computing* value r corresponding to argument d

some important notations

- λ -calculus (Church)
- Turing machines (Turing)
- Partial recursive functions



The λ -calculus: a simple type-free language

- to express *all computable functions*
- to directly express *higher-order functions*
- to study *evaluation orders, termination, uniqueness of answers...*
- to study various *typing systems*
- to serve as a *kernel language for functional languages*
 - However, λ -calculus extended with constants and let-blocks is more suitable

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

λ -notation

- a way of writing and applying functions without having to give them names
- a syntax for making a function expression from any other expression
- the syntax distinguishes between the *integer "2"* and the *function "always_two"* which when applied to any integer returns 2

`always_two x = 2;`

September 11, 2002

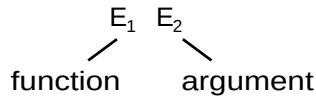
<http://www.csg.lcs.mit.edu/6.827>

Pure λ -calculus: Syntax

$$E = x \mid \lambda x. E \mid E E$$

variable abstraction application

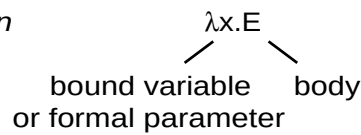
1. application



- application is left associative

$$E_1 E_2 E_3 E_4 \equiv (((E_1 E_2) E_3) E_4)$$

2. abstraction



- the scope of the dot in an abstraction extends as far to the right as possible

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

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Free and Bound Variables

- λ -calculus follows *lexical scoping* rules
- Free variables* of an expression

$$\begin{aligned} FV(x) &= \{x\} \\ FV(E_1 E_2) &= FV(E_1) \cup FV(E_2) \\ FV(\lambda x. E) &= FV(E) - \{x\} \end{aligned}$$

- A variable occurrence which is not free in an expression is said to be a *bound variable* of the expression
- combinator*: a λ -expression without free variables,
aka *closed λ -expression*

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

β -substitution

$(\lambda x.E) E_a \rightarrow E[E_a/x]$
 replace all free occurrences of x in E with E_a

$E[A/x]$ is defined as follows by case on E :

variable

$y[E_a/x] = E_a$ if $x \equiv y$
 $y[E_a/x] = y$ otherwise

application

$(E_1 E_2)[E_a/x] = (E_1[E_a/x] E_2[E_a/x])$

abstraction

$(\lambda y.E_1)[E_a/x] = \lambda y.E_1$ if $x \equiv y$
 $(\lambda y.E_1)[E_a/x] = \lambda z.((E_1[z/y])[E_a/x])$ otherwise
 where $z \notin FV(E_1) \cup FV(E_a) \cup FV(x)$

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>


β -substitution: an example

$(\lambda p.p (p q)) [(a p b) / q]$

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>


λ -Calculus as a Reduction System

Syntax

$$E = x \mid \lambda x.E \mid E E$$

Reduction Rule

$$\alpha\text{-rule: } \lambda x.E \rightarrow \lambda y.E [y/x] \quad \text{if } y \notin FV(E)$$

$$\beta\text{-rule: } (\lambda x.E) E_a \rightarrow E [E_a/x]$$

$$\eta\text{-rule: } (\lambda x.E x) \rightarrow E \quad \text{if } x \notin FV(E)$$

Redex

$$(\lambda x.E) E_a$$

Normal Form

An expression without redexes

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

α and η Rules

α -rule says that the bound variables can be renamed systematically:

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

η -rule can turn any expression, including a constant, into a function:

$$\lambda x.a x \rightarrow_{\eta} a$$

η -rule does not work in the presence of types

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

A Sample Reduction

$$\begin{aligned} C &\equiv \lambda x. \lambda y. \lambda f. f \ x \ y \\ H &\equiv \lambda f. f \ (\lambda x. \lambda y. x) \\ T &\equiv \lambda f. f \ (\lambda x. \lambda y. y) \end{aligned}$$

What is $H \ (C \ a \ b)$?

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>


Integers: Church's Representation

$$\begin{aligned} 0 &\equiv \lambda x. \lambda y. y \\ 1 &\equiv \lambda x. \lambda y. x \ y \\ 2 &\equiv \lambda x. \lambda y. x \ (x \ y) \\ \dots \\ n &\equiv \lambda x. \lambda y. x \ (x \dots (x \ y) \dots) \end{aligned}$$

succ ?

If n is an integer, then $(n \ a \ b)$ gives n nested a 's followed by b

$\Rightarrow$ the successor of n should be $a \ (n \ a \ b)$

$$\begin{aligned} \text{succ} &\equiv \lambda n. \lambda a. \lambda b. a \ (n \ a \ b) \\ \text{plus} &\equiv \lambda m. \lambda n. m \ \text{succ} \ n \\ \text{mul} &\equiv \end{aligned}$$

?

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>


Booleans and Conditionals

True $\equiv \lambda x. \lambda y. x$

False $\equiv \lambda x. \lambda y. y$

zero? $\equiv \lambda n. n (\lambda y. \text{False}) \text{True}$

zero? 0 $\rightarrow$

?

zero? 1 $\rightarrow$

?

cond $\equiv \lambda b. \lambda x. \lambda y. b \times y$

cond True $E_1 E_2 \rightarrow$

?

cond False $E_1 E_2 \rightarrow$

?

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>


Recursion ?

**fact n = if (n == 0) then 1
 else n * fact (n-1)**

- Assuming suitable combinators, fact can be rewritten as:

fact = $\lambda n. \text{cond (zero? n) 1 (mul n (fact (sub n 1)))}$

- How do we get rid of the fact on the RHS?*

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>


Choosing Redexes

$$1. \quad \begin{array}{cc} ((\lambda x.M) A) & ((\lambda x.N) B) \\ \text{-----} \rho_1 & \text{-----} \rho_2 \end{array}$$

$$2. \quad \begin{array}{c} ((\lambda x.M) ((\lambda y.N) B)) \\ \text{-----} \rho_2 \\ \text{-----} \rho_1 \end{array}$$

Does ρ_1 followed by ρ_2 produce the same expression as ρ_2 followed by ρ_1 ?

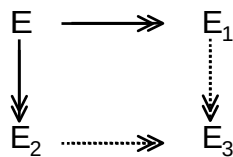
Notice in the second example ρ_1 can *destroy* or *duplicate* ρ_2 .

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Church-Rosser Property

A reduction system is said to have the *Church-Rosser property*, if $E \rightarrow E_1$ and $E \rightarrow E_2$ then there exists a E_3 such that $E_1 \rightarrow E_3$ and $E_2 \rightarrow E_3$.



also known as *CR* or *Confluence*

Theorem: The λ -calculus is CR.
(Martin-Lof & Tate)

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Interpreters

An *interpreter* for the λ -calculus is a program to reduce λ -expressions to “answers”.

It requires:

- the definition of an *answer*
- a *reduction strategy*
 - a method to choose redexes in an expression
- a criterion for *terminating* the reduction process

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Definitions of “Answers”

- *Normal form (NF)*: an expression without redexes
- *Head normal form (HNF)*:
 - x is HNF
 - $(\lambda x.E)$ is in HNF if E is in HNF
 - $(x E_1 \dots E_n)$ is in HNF
 - Semantically most interesting- represents the information content of an expression
- *Weak head normal form (WHNF)*:
 - An expression in which the left most application is not a redex.
 - x is in WHNF
 - $(\lambda x.E)$ is in WHNF
 - $(x E_1 \dots E_n)$ is in WHNF
 - Practically most interesting $\Rightarrow$ “Printable Answers”

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Reduction Strategies

There are many methods of choosing redexes in an expression

$$\begin{array}{c}
 ((\lambda x.M) ((\lambda y.N)B)) \\
 \text{----- } \rho_2 \text{-----} \\
 \text{----- } \rho_1 \text{-----}
 \end{array}$$

- *applicative order*: left-most innermost redex
- would reduce ρ_2 before ρ_1
- *normal order*: left-most (outermost) redex
- would reduce ρ_1 before ρ_2

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Facts

1. Every λ -expression does not have an answer
i.e., a NF or HNF or WHNF

$$\begin{array}{c}
 (\lambda x. x \ x) (\lambda x. x \ x) = \Omega \\
 \Omega \rightarrow
 \end{array}$$

2. CR implies that if NF exists it is *unique*

3. Even if an expression has an answer, not all *reduction strategies* may produce it

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

$$\text{leftmost redex: } (\lambda x. \lambda y. y) \ \Omega \rightarrow \lambda y. y$$

$$\text{innermost redex: } (\lambda x. \lambda y. y) \ \Omega \rightarrow$$

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

Normalizing Strategy

A *reduction strategy* is said to be *normalizing* if it terminates and produces an answer of an expression whenever the expression has an answer.

aka *the standard reduction*

Theorem: Normal order (left-most) reduction strategy is normalizing for the λ -calculus.

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

A Call-by-name Interpreter

Answers: WHNF

Strategy: leftmost redex

$cn(E)$: Definition by cases on E

$E = x \mid \lambda x. E \mid E E$

$cn(x) = x$

$cn(\lambda x. E) = \lambda x. E$

$cn(E_1 E_2) = \text{let } f = cn(E_1) \text{ in}$

case f of

$\lambda x. E_3 = cn(E_3[E_2/x])$

$_ = (f E_2)$

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>

A Call-by-value Interpreter

Answers: WHNF

Strategy: leftmost-innermost redex but not inside a λ -abstraction

cv(E): Definition by cases on E

$E = x \mid \lambda x. E \mid E E$

$cv(x) = x$

$cv(\lambda x. E) = \lambda x. E$

$cv(E_1 E_2) = \text{let } f = cv(E_1)$
 $\quad \quad \quad a = cv(E_2)$

in

case f of

$\lambda x. E_3 = cv(E_3[a/x])$
 $\quad \quad \quad = (f a)$

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>


More Facts

For computing WHNF

the call-by-name interpreter is normalizing
 but the call-by-value interpreter is not

e.g.

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

September 11, 2002

<http://www.csg.lcs.mit.edu/6.827>
