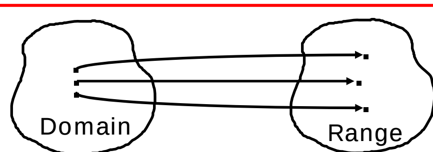
λ-calculus:
A Basis for Functional Languages

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

September 14, 2006

Functions



$f : D \rightarrow R$

f may be viewed as

- a set of ordered pairs < *d* , *r* > where $d \varepsilon D$ and $r \varepsilon R$
- 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

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;
```

Pure λ-calculus: Syntax

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

variable abstraction application

1. *application* $E_1\ E_2$

function argument

- application is left associative

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

2. *abstraction* $\lambda x.E$

bound variable body
or formal parameter

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

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

Free and Bound Variables

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

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

$FV(E_1\ E_2) = \boxed{FV(E_1) \cup FV(E_2)}$?

$FV(\lambda x.E) = \boxed{FV(E) - \{x\}}$?

- 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*

ß-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 $x \equiv y$
$y[E_a/x] = \boxed{y}$ otherwise ?

application

$(E_1\ E_2)[E_a/x] = \boxed{(E_1[E_a/x]\quad E_2[E_a/x])}$?

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 $z \notin FV(E_1) \cup FV(E_a) \cup FV(x)$

ß-substitution: an example

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

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

$\rightarrow\quad (\lambda z.z\ \ (z\ \ (a\ p\ b)))$

λ-Calculus as a Reduction System

Syntax

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

Reduction Rule

α-rule: $\lambda x.E \rightarrow \lambda y.E\ [y/x]$ if $y \notin FV(E)$

β-rule: $(\lambda x.E)\ E_a \rightarrow E\ [E_a/x]$

η-rule: $(\lambda x.E\ x) \rightarrow E$ if $x \notin FV(E)$

Redex

$(\lambda x.E)\ E_a$

Normal Form

An expression without redexes

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

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) ?

$$
\begin{aligned}
&\rightarrow (\lambda f.f\ (\lambda x.\lambda y.x))\ (C\ a\ b) \\
&\rightarrow (C\ a\ b)\ \ (\lambda x.\lambda y.x) \\
&\rightarrow (\lambda x.\lambda y.x)\ a\ b \\
&\rightarrow (\lambda y.a)\ b \\
&\rightarrow a
\end{aligned}
$$

$$
\begin{aligned}
H\ (C\ a\ b) &\twoheadrightarrow a \\
T\ (C\ a\ b) &\twoheadrightarrow b
\end{aligned}
$$

Integers: Church's Representation

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

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)*

$$
\begin{aligned}
\text{succ} &\equiv \lambda n.\lambda a.\lambda b.a\ (n\ a\ \ b) \\
\text{plus} &\equiv \lambda m.\lambda n.m\ \text{succ}\ n \quad ? \\
\text{mul} &\equiv \lambda m.\lambda n.m\ (\text{plus}\ n)\ 0 \quad ?
\end{aligned}
$$

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 $\twoheadrightarrow (\lambda x.\lambda y.y)\ (\lambda y.\text{False})\ \text{True}$?
$\rightarrow (\lambda y.\ y)\ \text{True}$
$\rightarrow \text{True}$

zero? 1 $\twoheadrightarrow (\lambda x.\lambda y.x\ y)\ (\lambda y.\text{False})\ \text{True}$?
$\rightarrow (\lambda y.\text{False})\ \text{True}$
$\rightarrow \text{False}$

cond $\equiv \lambda b.\lambda x.\lambda y.\ b\ x\ y$
cond True $E_1\ E_2 \twoheadrightarrow E_1$?
cond False $E_1\ E_2 \twoheadrightarrow E_2$?

Recursion ?

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

- Assuming suitable combinators, fact can be rewritten as:

  fact = λn. cond (zero? n) 1 (mul n (fact (sub n 1)))

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

  Suppose

  H = λf.λn.cond (zero? n) 1 (mul n (f (sub n 1)))

  then fact = H fact

  --- fact is a solution of this equation???

  more on recursion in the next lecture

Choosing Redexes

1. $\underbrace{((\lambda x.M)\ A)}_{\rho_1}\ \underbrace{((\lambda x.N)\ B)}_{\rho_2}$

2. $\underbrace{((\lambda x.M)\ \underbrace{((\lambda y.N)B)}_{\rho_2})}_{\rho_1}$

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 .

Church-Rosser Property

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

$$\begin{tikzcd} E \arrow[r, two heads] \arrow[d, two heads] & E_1 \arrow[d, two heads, dotted] \\ E_2 \arrow[r, two heads, dotted] & E_3 \end{tikzcd}$$

also known as *CR* or *Confluence*

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

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

Definitions of "Answers"

- *Normal form (NF):* an expression without redexes
- *Head normal form (HNF):*

  x is HNF
  (λx.E) is in HNF if E is in HNF
  (x E_1 ... 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
  (λx.E) is in WHNF
  (x E_1 ... E_n) is in WHNF

  Practically most interesting ⇒"Printable Answers"

Reduction Strategies

Two common strategies

- *applicative order:* left-most innermost redex
  aka call by value evaluation
- *normal order:* left-most (outermost) redex
  aka call by name evaluation

$$\underbrace{(\lambda x.y)\ ((\underbrace{\lambda x.x\ x)\ (\lambda x.x\ x}_{\rho_2}))}_{\rho_1}$$

← applicative order (ρ_2)
← normal order (ρ_1)

Facts

1. Every λ-expression does not have an answer
   i.e., a NF or HNF or WHNF
   $(\lambda x.x\ x)\ (\lambda x.x\ x) = \Omega$
   $\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \ldots$
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
   $(\lambda x.\lambda y.y)\ \Omega$

   leftmost redex: $(\lambda x.\lambda y.y)\ \Omega \rightarrow \lambda y.y$
   innermost redex: $(\lambda x.\lambda y.y)\ \Omega \rightarrow (\lambda x.\lambda y.\ y)\ \Omega \rightarrow \ldots$

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.

A Call-by-name Interpreter

Answers: WHNF
Strategy: leftmost redex

Apply the function before evaluating the arguments

cn(E): Definition by cases on E

E = x | λx.E | E E

cn(x) = x
cn(λx.E) = λx.E
cn(E_1 E_2) = *let* f = cn(E_1)
in
case f *of*
$\lambda x.E_3$ = cn($E_3[E_2/x]$)
_ = (f E_2)

Better syntax ...

[[....]] represents syntax

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

cn([[x]]) = x
cn([[λx.E]]) = λx.E
cn([[E_1 E_2]]) = *let* f = cn([[E_1]])
in
case f *of*
[[λx.E_3]] = cn(E3[E2/x])
_ = (f E_2)

Meta syntax

still messy

A Call-by-value Interpreter

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

cv(E): Definition by cases on E

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

Evaluate the argument before applying the function

cv(x) = x
cv(λx.E) = λx.E
cv(E_1 E_2) = *let* f = cv(E_1)
a = cv(E_2)
in
case f *of*
λx.E_3 = cv(E_3[a/x])
_ = (f a)

Normalizing?

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

call by value reduction → $(\lambda x.y)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$

call by name reduction → y

Which interpreters (if any) are normalizing for computing WHNF ?

call-by-value	*Clearly not*
call-by-name	*May be*

The proof to show that the call-by-name interpreter is normalizing is non-trivial

Big Step Semantics

- Consider the following rule

$$\frac{E_1 \Rightarrow \lambda x.E_b}{E_1\ E_2 \Rightarrow E_b\ [E_2 / x]}$$

- Can we compute using this rule?
- What does it compute?
- Will it compute every thing that the –calculus can?