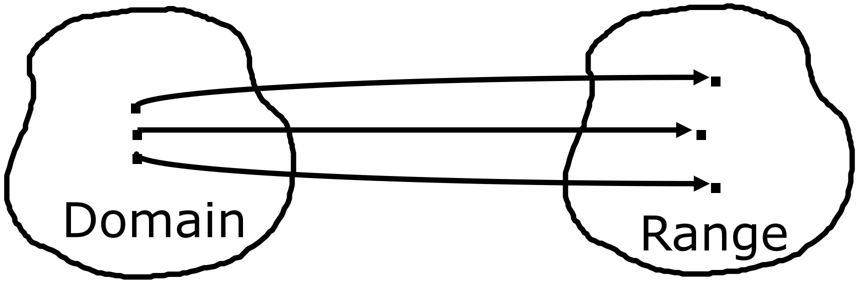
The λ-calculus

Armando Solar Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Adapted from Arvind 2010.
Used with permission.

Functions



$f : D \rightarrow R$

- A function may be viewed as a set of ordered pairs <*d*,*r*> where $d \in D$ and $r \in R$
- But we need to specify a *method of computing* value r corresponding to argument d
- Important notations for this purpose were developed in the 1930s
  - λ-calculus (Church)
  - Turing machines (Turing)
  - Partial recursive functions

The λ-calculus: a simple type-free language

- a way of *writing and applying functions without having to give them names*
- useful for studying *evaluation orders*, *termination*, *confluence*...
- useful for studying various *typing systems*
- often serves as *a kernel language for functional languages*

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 or formal parameter body

- 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

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

- A variable occurrence which is not free in an expression is said to be a *bound variable* of the expression

combinator or closed λ-expression: a λ-expression without free variables

β-substitution

$$\boxed{\begin{array}{c}(\lambda x.E)\ E_a \rightarrow E[E_a/x] \\ \text{replace all free occurrences of x in E with } E_a\end{array}}$$

E[A/x] is defined by cases on E:

variable

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

$y[E_a/x] = \boxed{y} \quad$ otherwise ?

application

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

abstraction

$(\lambda y.E_1)[E_a/x] = \lambda y.E_1 \quad$ if $x \equiv y$

$(\lambda y.E_1)[E_a/x] = \boxed{\begin{array}{l}\lambda z.((E_1[z/y])[E_a/x]) \quad \text{otherwise} \\ \quad \text{where } z \notin FV(E_1) \cup FV(E_a) \cup FV(x)\end{array}}$

β-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:

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

η-rule does not work in the presence of types; we will not consider it any further

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) ?

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

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

Integers: Church's Representation

$0 \equiv \lambda x.\lambda y.\ y$
$1 \equiv \lambda x.\lambda y.\ x\ y$
$2 \equiv \lambda x.\lambda y.\ x\ (x\ y)$
...
$n \equiv \lambda x.\lambda y.\ x\ (x\ldots(x\ y)\ldots)$

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 $\boxed{a\ (n\ a\ b)}$

succ $\equiv$ $\boxed{\lambda n.\lambda a.\lambda b.a\ (n\ a\ \ b)}$?

Integer Arithmetic

$0 \equiv \lambda x.\lambda y.\ y$
$1 \equiv \lambda x.\lambda y.\ x\ y$
$2 \equiv \lambda x.\lambda y.\ x\ (x\ y)$
...
$n \equiv \lambda x.\lambda y.\ x\ (x \ldots (x\ y) \ldots)$

succ $\equiv$ $\lambda n.\lambda a.\lambda b.a\ (n\ a\ \ b)$

plus $\equiv$ $\lambda m.\lambda n.m$ succ n ?
plus m n -- apply succ m times to n

mul $\equiv$ $\lambda m.\lambda n.m$ (plus n) 0 ?
mul m n -- apply (plus n) to 0 m times

Booleans and Conditionals

$$\text{True} \equiv \lambda x.\lambda y.x$$
$$\text{False} \equiv \lambda x.\lambda y.y$$

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

$$\text{zero? } 0 \twoheadrightarrow (\lambda x.\lambda y.y)\ (\lambda y.\text{False})\ \text{True} \quad ?$$
$$\rightarrow (\lambda y.\ y)\ \text{True}$$
$$\rightarrow \text{True}$$

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

$$\text{cond} \equiv \lambda b.\lambda x.\lambda y.\ b\ x\ y$$
$$\text{cond True}\ E_1\ E_2 \twoheadrightarrow E_1 \quad ?$$
$$\text{cond False}\ E_1\ E_2 \twoheadrightarrow E_2 \quad ?$$

Meaning of a term

- The semantics must distinguish between terms that should not be equal, i.e., if

  $0 \equiv \lambda x.\lambda y.\ y;\quad 1 \equiv \lambda x.\lambda y.\ x\ y;\quad 2 \equiv \lambda x.\lambda y.\ x\ (x\ y)$

  Then $\lambda x.\lambda y.\ y \neq \lambda x.\lambda y.\ x\ y \neq \lambda x.\lambda y.\ x\ (x\ y)$
- The semantics must equate terms that should be equal, i.e., the terms corresponding to (plus 0 1) and 1 must have the same meaning
- A semantics is said to be *fully abstract* if two terms have different meaning according to the semantics then there exists a term that can tell them apart

In the λ-calculus it is possible to define the meaning of a term almost syntactically