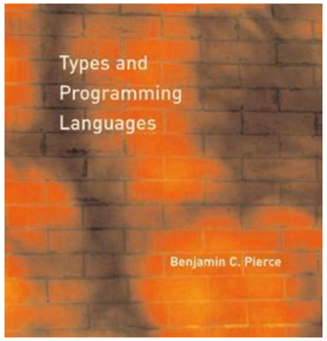
Lambda Calculus

UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA
DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on Chapter 5 of:
Benjamin C. Pierce, Types and Programming Languages. MIT Press, 2002.



Outline

- Syntax of the lambda calculus
  - abstraction over variables
- Operational semantics
  - beta reduction
  - substitution
- Programming in the lambda calculus
  - representation tricks
- Operational semantics of the lambda calculus
  - substitution
  - alpha-conversion, beta reduction
  - evaluation

Basic ideas

- introduce variables ranging over values
- define functions by (lambda-) abstracting over variables
- apply functions to values

$$x + 1$$

$$\lambda x.\ x + 1$$

$$(\lambda x.\ x + 1)\ 2$$

Abstract syntax

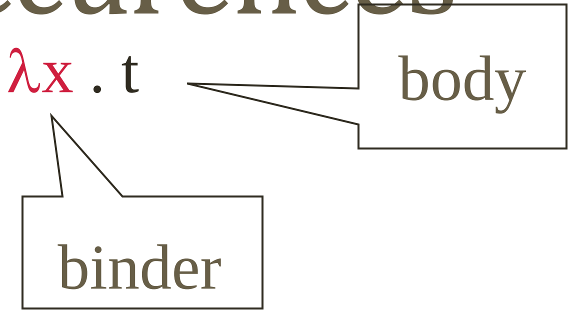
Pure lambda calculus: start with *nothing but variables*.

Lambda terms

t ::=

x	variable
λx . t	abstraction
t t	application

Scope, free and bound occurences



Occurences of x in the body t are bound. Nonbound variable occurrences are called free.

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

Beta reduction

Computation in the lambda calculus takes the form of beta-reduction:

$(\lambda x.\ t_1)\ t_2 \rightarrow [x \Rightarrow t_2]t_1$

where $[x \Rightarrow t_2]t_1$ denotes the result of substituting t_2 for all free occurrences of x in t_1

A term of the form $(\lambda x.\ t_1)\ t_2$ is called a beta-redex (or β-redex).

A (beta) normal form is a term containing no beta-redexes.

Beta reduction: Examples

$\underline{(\lambda x.\lambda y.y\ x)(\lambda z.u)} \rightarrow \lambda y.y(\lambda z.u)$

$\underline{(\lambda x.\ x\ x)(\lambda z.u)} \rightarrow (\lambda z.u)\ (\lambda z.u)$

$(\lambda y.y\ a)(\underline{(\lambda x.\ x)(\lambda z.(\lambda u.u)\ z)}) \rightarrow (\lambda y.y\ a)(\lambda z.(\lambda u.u)\ z)$

$(\lambda y.y\ a)((\lambda x.\ x)(\lambda z.\underline{(\lambda u.u)\ z})) \rightarrow (\lambda y.y\ a)((\lambda x.\ x)(\lambda z.\ z))$

$\underline{(\lambda y.y\ a)((\lambda x.\ x)(\lambda z.(\lambda u.u)\ z))} \rightarrow ((\lambda x.\ x)(\lambda z.(\lambda u.u)\ z))\ a$

Evaluation strategies

- Full beta-reduction
  - any beta-redex can be reduced
- Normal order
  - reduce the leftmost-outermost redex
- Call by name
  - reduce the leftmost-outermost redex, but not inside abstractions
  - abstractions are normal forms
- Call by value
  - reduce leftmost-outermost redex where argument is a value
  - no reduction inside abstractions (abstractions are values)

Complete Beta-reducción

Any redex may be evaluated. For example

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

that can be rewritted as

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

contains three redexes:

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

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

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

We take the outer redex, the inner redex and then the outer redex:

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

Normal order

Left-most, outer redexes are evaluated first:

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

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

$\lambda\, z.\underline{\mathrm{id}\ z} \to$

$\lambda\, z.z$

Call by name

Similar to normal order, with additional restriction:
Reductions inside abstractions are NOT allowed

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

Used byAlgol-60 and Haskell (*call by need*).

Call by value

The outer redex is applied but a redex can be applied only when the term to its right has been reduced to a value (variable or abstraction)

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

Function arguments are always eveluated, even they are not going to be used in the body.

The most used strategy (ML, Lisp, C, Java,…)

Programming in the lambda calculus

- multiple parameters through currying
- booleans
- pairs
- Church numerals and arithmetic
- lists
- recursion
    - call by name and call by value versions

Abstract Syntax

- V is a countable set of variables
- T is the set of terms defined by

$$
\begin{array}{lll}
t ::= & x & (x \in V) \\
 & \mid\ \lambda x.t & (x \in V) \\
 & \mid\ t\ t &
\end{array}
$$

Free variables

The set of free variables of a term is defined by

$FV(x) = \{x\}$
$FV(\lambda x.t) = FV(t) \setminus \{x\}$
$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$

E.g. $FV(\lambda x.\ y(\lambda y.\ xyu)) = \{y,u\}$

Substitution and free variable capture

Define substitution naively by

$[x \Rightarrow s]x = s$
$[x \Rightarrow s]y = y \quad \text{if } y \neq x$
$[x \Rightarrow s](\lambda y.t) = (\lambda y.[x \Rightarrow s]t)$
$[x \Rightarrow s](t_1\ t_2) = ([x \Rightarrow s]t_1)\ ([x \Rightarrow s]t_2)$

Then

(1) $[x \Rightarrow y](\lambda x.x) = (\lambda x.[x \Rightarrow y]x) = (\lambda x.y)$ wrong!

(2) $[x \Rightarrow y](\lambda y.x) = (\lambda y.[x \Rightarrow y]x) = (\lambda y.y)$ wrong!

(1) only free occurrences should be repalced.
(2) illustrates free variable capture.

Renaming bound variables

The name of a bound variable does not matter. We can change bound variable names, as long as we avoid free variables in the body:

Thus $\lambda x.x = \lambda y.y$

but $\lambda x.y \neq \lambda y.y$.

Change of bound variable names is called α-conversion.

To avoid free variable capture during substitution, we change bound variable names as needed.

Substitution refined

Define substitution

$[x \Rightarrow s]x = s$

$[x \Rightarrow s]y = y$ *if* $y \neq x$

$[x \Rightarrow s](\lambda y.t) = (\lambda y.[x \Rightarrow s]t)$ *if* $y \neq x$ *and* $y \notin FV(s)$

$[x \Rightarrow s](t_1\ t_2) = ([x \Rightarrow s]t_1)\ ([x \Rightarrow s]t_2)$

When applying the rule for $[x \Rightarrow s](\lambda y.t)$, we change the bound variable y if necessary so that the side conditions are satisfied.

Substitution refined (2)

The rule

$[x \Rightarrow s](\lambda y.t) = (\lambda y.[x \Rightarrow s]t)$ *if* $y \neq x$ *and* $y \notin FV(s)$

could be replaced by

$[x \Rightarrow s](\lambda y.t) = (\lambda z.[x \Rightarrow s][y \Rightarrow z]t)$
where $z \notin FV(t)$ and $z \notin FV(s)$

Note that $(\lambda x.t)$ contains no free occurrences of x, so
$[x \Rightarrow s](\lambda x.t) = \lambda x.t$

Operational semantics (call by value)

Syntax:

$t ::=$		*Terms*
	x	$(x \in V)$
	$\lambda x.t$	$(x \in V)$
	$t\ t$	

$v ::= \lambda x.t$ *Values*

We could also regard variables as values:

$v ::= x \mid \lambda x.t$

Operational semantics: rules

$(\lambda x.t1)\ v2 \rightarrow [x \Rightarrow v2]\ t1$

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2}$$

$$\frac{t_2 \rightarrow t_2'}{v_1\ t_2 \rightarrow v_1\ t_2'}$$

- evaluate function before argument
- evaluate argument before applying function

Booleans

Syntax:

t :: = … *Terms*

  true

  false

  if t then t else t

v ::= … *Values*

  true

  false

Booleans: Evaluation

If true then t_2 else $t_3 \rightarrow t_2$ (E-IfTrue)

If false then t_2 else $t_3 \rightarrow t_3$ (E-IfFalse)

$$\frac{t_1 \rightarrow t_1'}{\text{If } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{If } t_1' \text{ then } t_2 \text{ else } t_3} \quad \text{(E-If)}$$

Arithmetic expressions

```
t :: = …          Terms
        0
        succ t
        pred t
        iszero t

v ::= …           Values
        nv

nv ::=            Numeric Values
         0
         succ nv
```

Arithmetic evaluation

$$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'}$$ (E-SUCC)

$\text{pred } 0 \rightarrow 0$ (E-PREDZERO

$\text{pred (succ } nv_1) \rightarrow nv_1$ (E-PREDSUCC)

$$\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'}$$ (E-PRED)

$\text{iszero } 0 \rightarrow \text{true}$ (E-ISZEROZERO

$\text{iscero (succ } nv_1) \rightarrow \text{false}$ (E-ISZEROSUCC)

$$\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'}$$ (E-ISZERO)

Recursion

We need recursion to get repetitive evaluation!!

We will use a fixed-point combinator

We explain it on the blackboard…

(please read carefully section 5.2 of Pierce's book)