λ

Lambda Calculus - Part II

Programmazione Funzionale
2023/2024
Università di Trento
Chiara Di Francescomarino

Next lectures

- No class Thursday May 9
- Extra lecture on Monday May 20 11:30 – 13:30 Aula PC B106

Today

- Recap
- Beta-reductions
- Encodings

Recap

The lambda calculus

- Originally, the lambda calculus was developed as a logic by Alonzo Church in 1932
  - Church says: “There may, indeed, be other applications of the system than its use as a logic.”

The λ-calculus

1. Introduces variables ranging over values – e.g., $x + 1$
2. Define functions by (lambda-)abstracting over variables –e.g., $\lambda x. x + 1$
3. Apply functions to values – e.g., $(\lambda x. x + 1)2$

For instance we can write a function (computing the square of a variable) without naming it

$$(\lambda x. x^2)$$

and we can apply the function to another expression

$$(\lambda x. x^2)7 = 49$$

Formally

- When dealing with λ-calculus, given a countable set of variables V, we have

$$e ::= \; x \mid \lambda x.e \mid e\; e$$

that is, an expression e can be

- x: a variable $\in V$
- $\lambda x.e$: a function taking as input a parameter x and evaluating the expression e (abstraction)
- $e\; e$: the application of two expressions

Lambda calculus and ML syntax

λ-Calculus syntax

- $\lambda x. e$
- x: bound variable
- e: expression

ML syntax

- `fn x => e`
- `x`: formal parameter
- `e`: expression usually using `x`

Terminology

binder

$$\lambda x. e$$

The scope of x is the term e – body of the function

x is bound

$$\lambda x. xy$$

y is free

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

Conventions

- Associativity of application is on the left (as in ML)

  $y\ z\ x$ corresponds to $(y\ z)x$

- Parenthesis can be used for readability – though not strictly needed
  - $(((f_1 f_2) f_3) f_4)$ is more clear than $f_1 f_2 f_3 f_4$
- The body of a lambda extends as far as possible to the right, that is

  $\lambda x.\, x\ \lambda z.\, x\ z\ x$ corresponds to $\lambda x.\left(x\ \lambda z.\, (x\ z\ x)\right)$ and not to $\cancel{(\lambda x.\, x)\ (\lambda z.\, (x\ z\ x))}$

- Consecutive abstractions can be uncurried:

$$\lambda xyz.\, e = \lambda x.\, \lambda y.\, \lambda z.\, e$$

Free and bound variables

- The set of free variables of an expression is defined by:
  - $F_v(x) = \{x\}$
  - $F_v(\lambda x. e) = F_v(e) \backslash \{x\}$
  - $F_v(e_1 e_2) = F_v(e_1) \cup F_v(e_2)$

  e.g., $F_v(\lambda x. y(\lambda y. xyu)) = \{y, u\}$
- The set of bound variables of an expression is defined by
  - $B_v(x) = \emptyset$
  - $B_v(\lambda x. e) = \{x\} \cup B_v(e)$
  - $B_v(e_1 e_2) = B_v(e_1) \cup B_v(e_2)$

  e.g., $B_v(\lambda x. y(\lambda y. xyu)) = \{x, y\}$

Exercise 7.1

- Make the parentheses explicit in the following λ-expression

$$(\lambda p.\, pz)\lambda q.\, w\lambda w.\, wqzp$$

Solution exercise 7.1

- Make the parentheses explicit in the following λ-expression

$$(\lambda p. pz)(\lambda q. w(\lambda w. (((wq)z)p)))$$

Exercise 7.2

- In the following expression say which, if any, variables are bound (and to which λ), and which are free:

$$\lambda s.\, sz\lambda q.\, sq$$

Solution exercise 7.2

- In the following expression say which, if any, variables are bound (and to which λ), and which are free.

$$\lambda s.\, sz\lambda q.\, sq$$

- Both occurrences of s are bound to the first λ
- z is free
- q is bound to the second λ

Exercise 7.3

- In the following expression say which, if any, variables are bound (and to which λ), and which are free:

$$(\lambda s.\, sz)\lambda q.\, w\lambda w.\, wqzs$$

Solution exercise 7.3

- In the following expression say which, if any, variables are bound (and to which λ), and which are free:

$$(\lambda s.\, sz)\lambda q.\, w\lambda w.\, wqzs$$

- s: first occurrence bound to the first λ, second occurrence free
- z: both occurrences free
- q: bound to the second λ
- w: first occurrence free, second one bound to the third λ

The intuition

- Consider this lambda expression:

$$(\lambda x. x + 1)4$$

It means that we apply the lambda abstraction to the argument 4, as if we apply the increment function to the argument 4.

- How do we do it?

  The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which bound occurrences of the formal parameter in the body are replaced with copies of the argument.

- This means: $(\lambda x. x + 1)4 \underset{\beta}{\rightarrow} 4 + 1$

β-reduction examples

- $(\lambda x. x + x)5 \rightarrow 5 + 5 \longrightarrow 10$
- $(\lambda x. 3)5 \rightarrow 3$

Parameters

- formal
- formal occurrence
- actual

It looks like we instantiate the formal parameter (i.e., the occurrences of the bound variable) with the actual parameter (the expression to which we are applying the function)

Beta-reduction

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

$$(\lambda x. e_1)e_2 \rightarrow e_1[e_2/x]$$

where $e_1[e_2/x]$ denotes the result of substituting e_2 for all free occurrences of x in e_1.

- A term of the form $(\lambda x. e_1)e_2$ (that is an application with an abstraction on the left) is called beta-redex (or β-redex).
- A (beta) normal form is a term containing no beta-redexes

Substitution

- $e_1[e_2/x]$: in expression e_1, replace every occurrence of x by e_2
- The result of the substitution is written with $\mapsto$
- A simple example

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

- Three cases – the expression *e* is a(n):
  1. value
  2. application and
  3. abstraction

1. substitution in case of a value

- In $(\lambda x. e_1)e_2 \mapsto e_1[e_2/x]$, where e_1 is a value
  - If $e_1 = x, x[e_2/x] = e_2$
  - If $e_1 = y (\neq x), y[e_2/x] = y$

2. Substitution in case of application

- In $(\lambda x.e_1)e_2 \mapsto e_1[e_2/x]$, where e_1 is an application $e_{11}e_{12}$

$(e_{11}e_{12})[e_2/x]=(e_{11}[e_2/x]e_{12}[e_2/x])$

3. substitution in case of abstraction

- In $(\lambda x. e_1)e_2 \mapsto e_1[e_2/x]$, where e_1 is an abstraction $\lambda y. e$
  - If $y \neq x$ and $y \notin F_v(e_2)$, then
    $$(\lambda y. e)[e_2/x] = \lambda y. e[e_2/x]$$
  - If $y = x$, then
    $$(\lambda y. e)[e_2/x] = \lambda y. e$$

There is no effect of the substitution

- What happens instead if $y \in F_v(e_2)$?
  - We need to be careful!

Variable capture

- What happens when $y \in F_v(e_2)$?
- For instance what happens with $(\lambda x. \lambda y. x\ y) y$?
- When we replace y inside the expression, we do not want to be captured by the inner binding of y (it would violate the static scoping), that is, if we apply $(\lambda y. e)[e_2/x]$=$\lambda y. e[e_2/x]$, we would get
  $(\lambda y. x\ y)[y/x] \mapsto \lambda y. (x\ y[y/x]) = \lambda y. yy$ but
  $(\lambda x. \lambda y. x\ y)y \neq \lambda y. yy$
- Solution: rename y in v, that is change $\lambda y. x\ y$ to $\lambda v. x\ v$

$$(\lambda v. x\ v)[y/x] \mapsto \lambda v. (x\ v[y/x]) = \lambda v. yv$$

An example

```
int x=0;
int foo (name int y){
    int x = 2;
    return x + y;
}
...
int a = foo(x+1);
```

- Blindly applying the copy rule would lead us to a result of x+x+1=5
- Incorrect result as it would depend on the name of the local variable
- With a body `{int z = 2; return z + y;}` the result would have been z+x+1=3

- When the body contains the same name of the actual parameter, we say that it is captured by the local declaration
- In order to avoid substitutions in which the actual parameter is captured by the local declaration, we impose that the formal parameter – even after the substitution – is evaluated in the environment of the caller and not of the callee

Equivalence

- Given two expressions e_1 and e_2, when should they be considered to be equivalent?
    - Natural answer: when they differ only in the names of the bound variables
- If y is not present in e,

$$\lambda x. e \equiv \lambda y. e[y/x]$$

- This is called α −equivalence
- Two expressions are α −equivalent if one can be obtained from the other by replacing part of one by an α −equivalent one

α-Conversion

- α-conversion can be used to avoid having variable capture during substitution
- Examples

$$\lambda x.\, x =_{\alpha} \lambda y.\, y$$
$$\lambda x.\, xy =_{\alpha} \lambda z.\, zy$$

- But NOT

$$\lambda y.\, xy =_{\alpha} \lambda y.\, zy$$

3. substitution in case of abstraction

- In $(\lambda x. e_1)e_2 \mapsto e_1[e_2/x]$, where e_1 is an abstraction $\lambda y. e$
  - If $y \neq x$ and $y \notin F_v(e_2)$, then

    $$(\lambda y. e)[e_2/x] = \lambda y. e[e_2/x]$$
  - If $y = x$, then

    $$(\lambda y. e)[e_2/x] = \lambda y. e$$

    There is no effect of the substitution
  - What happens instead if $y \in F_v(e_2)$?
    - We need to be careful!
    - We have to rename the name of the formal parameter (so that it does not depend anymore on e_2). Indeed:
      - $\lambda y. y = \lambda z. z$
      - $\lambda y. e = \lambda z. (e[z/y])$

Question 4

Which of the following reduces to $\lambda z. z$?

$A.\ (\lambda y. \lambda z. x)\ z$
$B.\ (\lambda z. \lambda x. z)\ y$
$C.\ (\lambda y. y)(\lambda x. \lambda z. z)w$
$D.\ (\lambda y. \lambda x. z)z(\lambda z. z)$

Answer question 4

Which of the following reduces to $\lambda z. z$?

$A.\ (\lambda y. \lambda z. x)\ z$
$B.\ (\lambda z. \lambda x. z)\ y$
$C.\ (\lambda y. y)(\lambda x. \lambda z. z)w$
$D.\ (\lambda y. \lambda x. z)z(\lambda z. z)$

Question 5

Which of the following expressions is apha-equivalent to $(\lambda x. \lambda y. x\ y)y$?

$A.\ \lambda y. y\ y$
$B.\ \lambda z. y\ y$
C. $(\lambda x. \lambda z. x\ z)y$
D. $(\lambda x. \lambda y. x\ y)y$

Answer question 5

Which of the following expressions is alpha-equivalent to $(\lambda x.\lambda y.x\ y)y$?

A. $\lambda y.y\ y$
B. $\lambda z.y\ y$
C. $(\lambda x.\lambda z.x\ z)y$
D. $(\lambda x.\lambda y.x\ y)y$

Question 6

Beta-reducing the following term produces what result? $\lambda x.\,(\lambda y.\, y\ y) w\ z$

A. $\lambda x.\, w\ w\ z$
B. $\lambda x.\, w\ z$
C. $w\ z$
D. Does not reduce

Answer question 6

Beta-reducing the following term produces what result? $\lambda x.(\lambda y.y\ y)w\ z$

A. $\lambda x.w\ w\ z$
B. $x.w\ z$
C. $w\ z$
D. Does not reduce

Question 7

Beta-reducing the following term produces what result? $(\lambda x. x\ \lambda y. y\ x) y$

$A.\ y\ (\lambda z. z\ y)$
$B.\ z\ (\lambda y. y\ z)$
C. y $(\lambda y. y\ y)$
$D.\ y\ y$

Answer question 7

Beta-reducing the following term produces what result? $(\lambda x. x\ \lambda y. y\ x) y$

A. $y\ (\lambda z. z\ y)$
B. $z\ (\lambda y. y\ z)$
C. y $(\lambda y. y\ y)$
D. $y\ y$

Few rules/guidelines … to remember for β-reduction

1. Associativity of applications is on the left: $M\ N\ L\ \equiv\ (M\ N)\ L$
2. The body of a lambda extends as far as possible to the right, e.g.,
   $\lambda x.x\ \lambda z.x\ z\ x$ corresponds to $\lambda x.\big(x\ \lambda z.(x\ z\ x)\big)$ and not to $\cancel{(\lambda x.x)\ (\lambda z.(x\ z\ x))}$
3. Consider the precedence rules imposed by parentheses – when they are used
4. Otherwise, precedence is given to the leftmost and innermost precedence, e.g.,
   $\big((\lambda x.x)x\big)(\lambda x.xy) \mapsto x(\lambda x.xy)$, while $\big((\lambda x.x)x\big)(\lambda x.xy) \mapsto \cancel{(\lambda x.xy)x}$ is incorrect!

Few rules/guidelines ... to remember for β-reduction

5. Be careful when a variable is captured (i.e., when a free variable becomes bound): this is an error! E.g.,
   $(\lambda y.(\lambda x.yx))x \nrightarrow (\lambda x.xx)$ as the free variable y becomes bound after the application ... we need to rename the bound x with a different name, e.g., t:
   $(\lambda y.(\lambda t.yt))x$, so as to avoid that variables are captured

You can find lambda functions …

- In ML

```
val square = fn x => x*x;
```

- In Python:

```
square = lambda x: x*x
```

Exercise 7.4

- Reduce to normal form
  - $(\lambda x. x(xy))(\lambda z. zx)$

Solution exercise 7.4

- Reduce to normal form
  - $(\lambda x. x(xy))(\lambda z. zx)$

    $(\lambda x. x(xy))(\lambda z. zx) \mapsto$

    $(\lambda z. zx)\,((\lambda z. zx)y) \mapsto$

    $(\lambda z. zx)\,(yx) \mapsto$

    $(yx)x$

Exercise 7.5

- Reduce to normal form
  - $(\lambda x. xy)(\lambda z. zx)(\lambda z. zx)$

Solution exercise 7.5

- Reduce to normal form
  - $(\lambda x.xy)(\lambda z.zx)(\lambda z.zx)$

    $(\lambda x.xy)(\lambda z.zx)(\lambda z.zx) \mapsto$

    $((\lambda z.zx)y)(\lambda z.zx) \mapsto$

    $(yx)(\lambda z.zx)$

Exercise 7.6

- Reduce to normal form
  - $(\lambda t.\, tx)((\lambda z.\, xz)(xz))$

Solution exercise 7.6

- Reduce to normal form
  - $(\lambda t.tx)\big((\lambda z.xz)(xz)\big)$

    $(\lambda t.tx)\big((\lambda z.xz)(xz)\big) \mapsto$

    $(\lambda t.tx)(x(xz)) \mapsto$

    $x(xz)x$

Higher-Order Functions

- Beta-reductions can be applied with higher-order functions
- For instance, given a function f, return function $f \circ f$

  $\lambda f.\ \lambda x.\ f\ (f\ x)$
- How does this work?

$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y + 1) \mapsto_\beta$$

$$(\lambda x.\ (\lambda y.\ y{+}1)\ ((\lambda y.\ y{+}1)\ x)) \mapsto_\beta$$

$$(\lambda x.\ (\lambda y.\ y{+}1)\ (x{+}1)) \mapsto_\beta$$

$$(\lambda x.\ (x{+}1){+}1)$$

Same result if executing first λy

Same Procedure (ML)

- Given function f, return function f ∘ f

```
fn f => fn x => f(f(x));
val it = fn: ('a -> 'a) -> 'a -> 'a
```

- How does this work?

```
(fn f => fn x => f(f(x))) (fn y => y + 1)

= fn x => ((fn y => y + 1) ((fn y => y + 1) x))

= fn x => ((fn y => y + 1) (x + 1))

= fn x => ((x + 1) + 1)
```

Same Procedure (JavaScript)

- Given function f, return function f ∘ f

```
function (f) { return function (x) { return f(f(x)); } ; }
```

- How does this work?

```
(function (f) { return function (x) { return f(f(x)); } ;  })
                       (function (y) { return y + 1; })


function (x) { return (function (y) { return y + 1; })
                              ((function (y) { return y + 1; }) (x)); }


function (x) { return (function (y) { return y +1; }) (x + 1); }


function (x) { return ((x + 1) + 1); }
```

Same Procedure (Python)

- Given function f, return function f ∘ f

```
def g(x): return (lambda f,x: f(f(x)))(lambda y:y+1,x)
```

- How does this work?

```
def g(x): return (lambda f,x: f(f(x)))(lambda y:y+1,x)

def g(x): return (lambda y:y+1,(lambda y:y+1,x))

def g(x): return (lambda y:y+1,(x + 1))

def g(x): return ((x + 1) + 1)
```

β −reductions

- β-reductions are not symmetric
- $e_1 \mapsto_\beta e_2$ does not imply $e_2 \mapsto_\beta e_1$
  - So this is not an equivalence relation
  - A notion of β-equivalence can be defined as the reflexive and transitive closure of $\mapsto_\beta$

Normal form

- Expressions with no redex, have no β-reductions
  - This is called normal form
  - $\lambda x. \lambda y. x$ is in normal form
  - $\lambda x. ((\lambda y. y) x)$ is not in normal form
  - $(\lambda y. y) x \;\mapsto_\beta\; x$ and therefore $\lambda x. (\lambda y. y) x \;\mapsto_\beta\; \lambda x. x$

Termination

- β-reductions may terminate in a normal form
- Or they may run forever

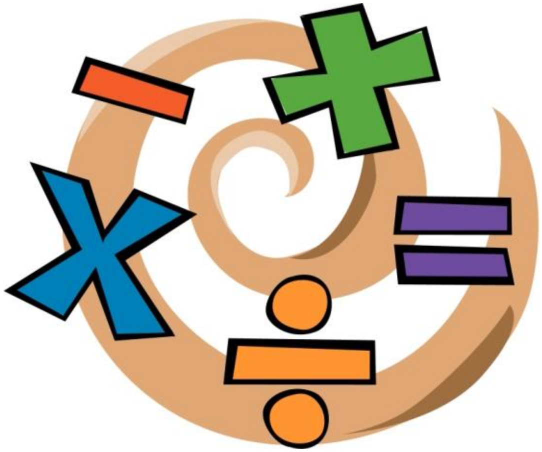
$$(\lambda x.\, xx)(\lambda x.\, xx) \;\mapsto_\beta\; (xx)([(\lambda x.\, xx)/x])$$
$$= (\lambda x.\, xx)(\lambda x.\, xx)$$

- This is similar to infinite recursion or infinite loops

Confluence

- Basic theorem

  If e can be reduced to e_1 by a β-reduction and e can be reduced to e_2 by a β-reduction, then there exists an e_3 such that both e_1 and e_2 can be reduced to e_3 by β-reductions
- This means that, if e can be reduced to a normal form, the order of the reductions does not matter



Encodings

The λ-calculus

- We have seen at the beginning a version of λ-calculus including constants (0,1,2) and functions (+,*)
- The pure λ-calculus, however, seems to be a very limited language
  - Expressions: Only variables, application and abstraction
  - For example, $\lambda x.x + 2$ should be invalid, since 2 is not a variable
- Despite this, the λ-calculus is very expressive
  - It is Turing-complete: Any computation can be expressed in the λ-calculus
  - We can encode any computations …
  - booleans, pairs, constants and arithmetic can be expressed

Booleans

- $true = \lambda x. \lambda y. x$
- $false = \lambda x. \lambda y. y$
- If a then b else c = a b c

- Examples
  - If true then b else c = $(\lambda x. \lambda y. x) b\ c \rightarrow (\lambda y. b) c \rightarrow b$
  - If false then b else c = $(\lambda x. \lambda y. y) b\ c \rightarrow (\lambda y. y) c \rightarrow c$

Booleans

- Other Booleans operations
  - not = $\lambda x. x\ false\ true$
    - not x = if x then false else true
    - not true → $(\lambda x. x\ false\ true) true \rightarrow (true\ false\ true) \rightarrow false$
  - and = $\lambda x. \lambda y. x\ y\ false$
    - and x y = if x then y else false
  - or = $\lambda x. \lambda y. x\ true\ y$
    - or x y = if x then true else y
- Given these operations
  - Can build up a logical inference system

Question 8

What is the lambda-calculus encoding for xor x y ?

- xor true true = xor false false = false
- xor true false = xor false true = true

$A.\ \lambda x. x\ x\ y$
$B.\ \lambda x. x\ (\lambda y. y\ true\ false)\ y$
$C.\ \lambda x. x\ (\lambda y. y\ false\ true)\ y$
$D.\ \lambda x. \lambda y. y\ x\ y$

- $true = \lambda x. \lambda y. x$
- $false = \lambda x. \lambda y. y$
- If a then b else c = a b c
- not = $\lambda x. x\ false\ true$

Answer question 8

What is the lambda-calculus encoding for xor x y ?

- xor true true = xor false false = false
- xor true false = xor false true = true

$A.\ \lambda x.x\ x\ y$
$B.\ \lambda x.x\ (\lambda y.y\ true\ false)\ y$
$C.\ \lambda x.x\ (\lambda y.y\ false\ true)\ y$
$D.\ \lambda x.\lambda y.y\ x\ y$

- $true = \lambda x.\lambda y.x$
- $false = \lambda x.\lambda y.y$
- If a then b else c = a b c
- not = $\lambda x.x\ false\ true$

It is as if we write
If x then not y else y

Summary

- Beta-reductions
- Econdings

Readings

- Chapter 11 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill
- Few slides from the University of Maryland

Next time

- Signatures and structures