λ

Lambda Calculus - Part I

Programmazione Funzionale
2024/2025
Università di Trento
Chiara Di Francescomarino

When you have time

You can find the link also in Moodle!

Join this Wooclap event

1. Go to wooclap.com
2. Enter the event code in the top banner

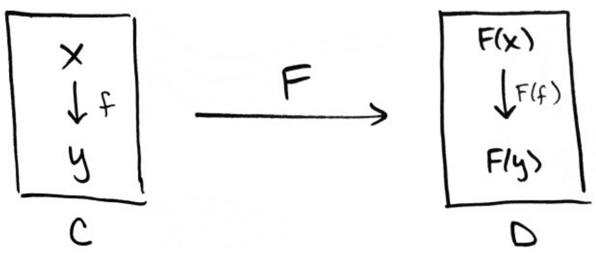
Event code
DQDQRX

Next lectures

- Extra-slot for tutoring on Friday May 11th (11:30 – 12:30)
- ML Challenge on Thursday May 15th
- One of the last classes, we will have the exam simulation

Today

- Functors in ML
- Lambda calculus
  - Introductory concepts
  - Beta-reductions
  - Alpha-equivalence



Functors in ML

Structures, signatures and functors

- Structures: Collections of type, datatypes, functions, exceptions, etc., that we want to encapsulate
- Signatures: Collections of information describing the types and other specifications for some of the elements of a structure
- :> : the structure has to implement what is defined in the signature; type implementation and methods not defined in the signature cannot be accessed

Signatures are used to hide the structure implementation

```
> signature MESSAGE = sig
    type message
    val createMessage : string -> message
end;

> structure StringMsgH :> MESSAGE = struct
  type message = string;
  fun createMessage s = s;
  fun printMessage s = print(s);
end;

> StringMsgH.createMessage "Hello world";
val it = ?: StringMsgH.message

> StringMsgH.printMessage "Hello world";
poly: : error: Value or constructor (printMessage) has not been declared in structure
StringMsgH
Found near StringMsgH.printMessage "Hello world"
Static Errors
```

Structures can be polymorphic in terms of types

```
> structure Mapping = struct
   exception NotFound;
   val create = nil;
   fun lookup (d,nil) = raise NotFound
     | lookup (d,(e,r)::es) =
         if d=e then r
         else lookup (d,es);
    fun insert (d,r,nil) = [(d,r)]
       | insert (d,r,(e,s)::es) =
            if d=e then (d,r)::es
            else (e,s)::insert(d,r,es)
end;

structure Mapping:
  sig
     exception NotFound
     val create: 'a list
     val insert: ''a * 'b * (''a * 'b) list -> (''a * 'b) list
     val lookup: ''a * (''a * 'b) list -> 'b
end
```

A structure that manipulates a list of key-value pairs (d,r). It has a create variable containing an empty list and allows for:

- Searching for the key and returning the value – if in the list
- Inserting a new pair

This is a polymorphic structure.

```
> Mapping.create;
val it = []: 'a list
```

Signatures can be used to restrict structure types

- Restrict mappings to be on string int pairs

```
signature SIMAPPING = sig
  val create : (string * int) list;
  val insert : string * int * (string * int) list -> (string * int) list;
  val lookup : string * (string * int) list -> int
end;

signature SIMAPPING =
  sig
   val create: (string * int) list
   val insert: string * int * (string * int) list -> (string * int) list
   val lookup: string * (string * int) list -> int
End

> structure SiMapping : SIMAPPING = Mapping;
structure SiMapping: SIMAPPING
```

Can we make structures parametric?

```
> structure BST = struct
   exception EmptyTree;
   datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree;
   fun lookup lt Empty x = false
       | lookup lt (Node(y,left,right)) x =
           if lt(x,y) then lookup lt left x
           else if lt(y,x) then lookup lt right x else true;
   fun insert lt Empty x = Node(x,Empty,Empty)
       |insert lt (T as Node (y,left,right)) x =
           if lt (x,y) then Node (y,(insert lt left x),right)
           else if lt (y,x) then Node (y,left,(insert lt right x)) else T;
   fun deletemin (Empty) = raise EmptyTree
           | deletemin (Node(y,Empty,right)) = (y,right)
           | deletemin (Node(w,left,right)) = let val (y,L) =
deletemin(left)
                                  in (y,Node(w,L,right))
                                  end;
   fun delete lt Empty x = Empty
      |delete lt (Node(y,left,right)) x =
           if lt (x,y) then Node(y,(delete lt left x),right)
           else if lt (y,x) then Node(y,left,(delete lt right x))
                else case (left,right) of (Empty,r) => r
                                  | (l,Empty) => l
                                  | (l,r) => let val (z,r1) = deletemin(r)
                                              in Node (z,l,r1)
                                  end;
end;
```

A structure that manipulates BSTs. Given a generic function `lt`, it allows for :

- Looking for x
- Inserting a node containing x
- Deleting the node containing x

Can we make structures parametric?

- Let us assume we want to apply the structure to string BSTs
  - We need to define an appropriate `lt`

```
> fun lower (nil) = nil
   | lower (c::cs) = (Char.toLower c)::lower (cs);
val lower = fn: char list -> char list
> fun lt (x,y) =  implode (lower (explode x)) < implode (lower (explode
y));
val lt = fn: string * string -> bool
```

  - We need to rewrite the structure

Can we make structures parametric?

```
> structure StringBST = struct
   exception EmptyTree;
   datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree;
   fun lookup Empty x = false
       | lookup (Node(y,left,right)) x =
            if lt(x,y) then lookup left x
            else if lt(y,x) then lookup right x else true;
   fun insert Empty x = Node(x,Empty,Empty)
       |insert (T as Node (y,left,right)) x =
            if lt (x,y) then Node (y,(insert left x),right
            else if lt (y,x)
                 then Node (y,left,(insert right x))
                 else T;
   fun deletemin (Empty) = raise EmptyTree
            | deletemin (Node(y,Empty,right)) = (y,right)
            | deletemin (Node(w,left,right)) =
              let val (y,L) = deletemin(left)
                       in (y,Node(w,L,right))
              end;
   fun delete Empty x = Empty
      |delete (Node(y,left,right)) x =
            if lt (x,y) then Node(y,(delete left x),right)
            else if lt (y,x) then Node(y,left,(delete right x))
                 else case (left,right) of (Empty,r) => r
                        | (l,Empty) => l
                        | (l,r) => let val (z,r1) = deletemin(r)
                                     in Node (z,l,r1)
                                end;
end;
```

```
structure StringBST:
  sig
    exception EmptyTree
    datatype 'a btree = Empty |
                  Node of 'a * 'a btree * 'a btree
    val delete: string btree -> string -> string btree
    val deletemin: 'a btree -> 'a * 'a btree
    val insert: string btree -> string -> string btree
    val lookup: string btree -> string -> bool
  end
```

Can we make structures parametric? - Issues

- We have strange types of the elements in the structure
- We do not create an object which is a BST that work on any type with an `lt` (less-then) function

Structures, signatures and functors

- Structures: Collections of type, datatypes, functions, exceptions, etc., that we want to encapsulate
- Signatures: Collections of information describing the types and other specifications for some of the elements of a structure
- :> : the structure has to implement what is defined in the signature; what is not defined in the signature cannot be accessed.
- Functors: Operations that take as arguments one or more elements such as structures, and produce a structure.

Functors

- Takes a structure and returns another structure
  - As a function takes a value and returns a new value a functor takes a structure and returns a new structure
- Consider our example of BSTs with comparison operator `lt`
- A functor takes as arguments a structure and a `less-than` operator and produces a structure incorporating the comparison operator

```
functor <identifier> (<structure name>:
<signature>) = <structure definition>
```

The steps we take

- Step 1: define a signature `TOTALORDER` that is satisfied by our functor inputs
- Step 2: define a functor `MakeBST` that takes a structure `S` with signature `TOTALORDER` and produces a structure
- Step 3: define a structure `StringOrder` with signature `TOTALORDER` and with a comparison operator on strings
- Step 4: apply `MakeBST` to `StringOrder` to produce the desired structure `StringBST`

Step 1: define the signature `TOTALORDER`

```
> signature TOTALORDER = sig
  type element;
  val lt : element * element -> bool
end;

signature TOTALORDER = sig type element val lt: element *
element -> bool end
```

Step 2: define the functor (sketch)

```
> functor MakeBST (Order: TOTALORDER):
  sig
    type 'label btree
    exception EmptyTeee;
    val create : Order.element btree;
    val lookup : Order.element * Order.element btree -> bool;
    val insert : Order.element * Order.element btree -> Order.element btree;
    val deletemin : Order.element btree -> Order.element Order.element btree;
    val delete : Order.element * Order.element btree -> Order.element btree
  end
=
  struct
   open Order;
   datatype 'label btree =
           Empty |
           Node of 'label * 'label btree * 'label btree;
    val create = Empty;
    val lookup (x, Empty) = ...
    val insert (x, Empty) = ...
    exception EmptyTree;
    fun deletemin (Empty) = ...
    fun delete (x,Empty) = ...
  end;
```

Step 3: define the functor argument

```
structure StringOrder:> TOTALORDER =
struct
   type element = string;
   fun lt (x,y) =
     let
        fun lower (nil) = nil
        | lower (c::cs) = (Char.toLower c)::lower (cs);
     in
        implode (lower (explode (x))) < implode (lower( explode
(y)))
      end;
end;
```

Step 4: Apply the functor

```
structure StringBST = MakeBST (StringOrder);
```

Extensions: applying functor to explicit structure

```
structure StringBST = MakeBST (
  struct
    type element = string;
    fun lt (x,y) =
       let
         fun lower (nil) = nil
         | lower (c::cs) = (Char.toLower c)::lower (cs);
       in
         implode (lower (explode (x))) < implode (lower(
explode (y)));
       end;
   end
);
```

λ

Lambda-calculus

What is the lambda-calculus?

- A very simple, but Turing complete, programming language
  - created before concept of *programming language* existed!
  - helped to define what *Turing complete* means!

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.”

Early history of the lambda calculus

- Meanwhile, in England …
  - young Alan Turing invents the Turing machine
- Turing heads to Princeton, studies under Church
  - prove lambda calculus, Turing machine, general recursion are equivalent – they define the class of computable functions
  - Church–Turing thesis: these capture all that can be computed

The λ-calculus

- Purpose: formal mathematical basis for functional programming
- Why lambda? Evolution of notation for a bound variable:
  - Whitehead and Russell, *Principia Mathematica*, 1910

    $2\hat{x} + 3$ – corresponds to $f(x) = 2x + 3$
  - Church's early handwritten papers

    $\hat{x}: 2x + 3$ – makes scope of variable explicit
  - Typesetter #1

    $\wedge x: 2x + 3$ – couldn't typeset the circumflex!
  - Typesetter #2

    $\lambda x. 2x + 3$ – picked a prettier symbol

Barendregt, *The Impact of the Lambda Calculus in Logic and Computer Science*, 1997

Impact of the lambda calculus

- Turing machine: theoretical foundation for imperative languages
  - Fortran, Pascal, C, C++, C#, Java, Python, Ruby, JavaScript, . . .
- Lambda calculus: theoretical foundation for functional languages
  - Lisp, ML, Haskell, OCaml, Scheme/Racket, Clojure, F#, Coq, . . .

The λ-calculus ingredients

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`

Two operations

- Abstraction of a term with respect to a variable x: $\lambda x. e$, that is the function that when applied to a value v produces e in which v replaces x
- Application of a function to an argument: $e_1 e_2$, that is the application of the function e_1 to the argument e_2

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

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

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

The abstraction (λ) operator removes a variable from the list of free variables and adds it to the bound ones

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:

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

Let’s try a test

Join this Wooclap event

1 Go to **wooclap.com**

2 Enter the event code in the top banner

Event code

PCEXEL

Exercise 10.1

- Make the parentheses explicit in the following λ-expression

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

Exercise 10.2

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

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

Exercise 10.3

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

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

β

Beta-reduction

Lambda expression evaluation

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)

β-reduction examples

- $\underbrace{(\lambda x.(\lambda y.y - x))\ 4\ 5}_{\text{We can see this as currying – we peel off the argument 4 and then 5}} \rightarrow (\lambda y.y - 4)5 \rightarrow 5 - 4 \rightarrow 1$

- $(\lambda f.f\ 3)(\lambda x.x + 1) \rightarrow (\lambda x.x + 1)3 \rightarrow 4$

Parameters
- formal
- formal occurrence
- actual

β-reduction examples

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

a curried function of two arguments: it applies its first argument to its second

Parameters

- formal
- formal occurrence
- actual

β-reduction examples

- $(\lambda x.\lambda y.x\ y)(\lambda z.zz)x \to (\lambda y.(\lambda z.zz)y)x$
  $\to (\lambda z.zz)x \to xx$
- $(\lambda x.x\ (\lambda y.y))(u\ r) \ \to (u\ r)(\lambda y.y)$
- $(\lambda x.(\lambda w.x\ w))(y\ z) \to \lambda w.(y\ z)w$

Parameters

- formal
- formal occurrence
- actual

β-reduction examples

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

$\rightarrow (\lambda x. (\lambda z. (x\ z))) y$ since λ extends to right

$\rightarrow (\lambda x. (\lambda z. (x\ z))) y$ apply $(\lambda x. e_1) e_2 \rightarrow e_1[e_2/x]$
where $e_1 = (\lambda z. (x\ z)), e_2 = y$

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

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
    $$(\lambda y. e)[e_2/x] = \lambda y. e$$
  - What happens instead if $y \in F_v(e_2)$?
    - We need to be careful!

There is no effect of the substitution

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
  $(\lambda y. x\ y)[y/x] \longmapsto \lambda y. (x\ y[y/x]) = \lambda y. yy$ 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 $\lambda v. x\ v$

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

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

There is no effect of the substitution

Let’s try a test

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
PCEXEL

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 expression extends as far as possible to the right, e.g.,
   $\lambda x.\, x\ \lambda z.\, x\ z\ x$ corresponds to $(\lambda x.\, (x\ (\lambda z.\, (x\ z\ x))))$ 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.,
   $((\lambda x.\, x)x)(\lambda x.\, xy) \mapsto x(\lambda x.\, xy)$, while $((\lambda x.\, x)x)(\lambda x.\, xy) \mapsto \cancel{(\lambda x.\, xy)x}$ 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 10.4

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

Exercise 10.5

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

Exercise 10.6

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

Summary

- Lambda calculus

Next time

- More on lambda calculus