λ

Curried functions and data types

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

Today

- Higher order functions
- Curried functions in ML
- Built-in higher order functions in ML
- Function composition in ML
- User-defined types in ML
- Data abstraction

When you have time

Join this Wooclap event

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

Event code
DQDQRX

Higher-order functions

Functions as parameters

Deep vs shallow binding

```
{int x = 1;
 int f(int y){
     return x+y;
 }
void g (int h(int b)){
     int x = 2;
     return h(3) + x;
}
...
 {int x = 4;
  int z = g(f);
 }
}
```

Shallow binding: environment at this point in time

Deep binding: environment at this point in time

- Deep binding: environment at the moment of creation of the link
- Shallow binding: environment at the moment of the call

- Dynamic scope
  - Possible with deep binding
  - Or shallow binding
- Static scope
  - Always uses deep binding

Functions as results

Functions as results

- Generating functions as the result of other functions allows the dynamic creation of functions at runtime

```
{int x = 1;
 void->int F () {
     int g () {
         return x+1;
     }
     return g;
 }
 void->int gg = F();
 int z = gg();
}
```

- **void-> int** denotes the type of the functions that take no argument and return an int
- **void->int** `F()` is the declaration of a function which returns a function of no argument and return value int
- **return g** returns the function and not its application
- gg is dynamically associated with the result of the evaluation of `F`
- The function gg returns the successor of the value of `x`
- `z` is finally equal to 2

Curried functions in ML

Curried functions

- Functions in ML have only one argument
- Functions with two arguments f(x,y) can be implemented as:
  - A function with a tuple as argument
  - Curried form
    - Unary function takes argument x
    - The result is a function $f(x)$ that takes argument y

- Curried function: divides its arguments such that they can be partially supplied producing intermediate functions that accept the remaining arguments

Example

```
> fun exponent1 (x,0) = 1.0
    | exponent1 (x,y) = x * exponent1 (x,y-1);
val exponent1 = fn: real * int -> real

> fun exponent2 x 0 = 1.0
    | exponent2 x y = x * exponent2 x (y-1);
val exponent2 = fn: real -> int -> real

> exponent1 (3.0,4);
val it = 81.0: real

> exponent2 3.0 4 ;
val it = 81.0: real
```

-> associates to right:
`real -> (int -> real)`
`exponent2` is a function taking a `real` and returning a function from `int` to `real`

Partial instantiation

- Curried functions are useful because they allow us to create partially instantiated or specialized functions where some (but not all) arguments are supplied.

```
> val g = exponent2 3.0;
val g = fn: int -> real

> g 4;
val it = 81.0: real

> g (4);
val it = 81.0: real
```

We are partially instantiating exponent2 (with name g) – g is the power function with base 3.0

Order of evaluation

- Parentheses are not necessary but we need to be careful as function application has the highest precedence
- `fun f c:char=1.0` means `(f c):char=1.0`. We probably mean `fun f(c:char)=1.0`
- fun f `x::xs=nil` means `(f x)::xs=nil`. We probably mean `fun f (x::xs)=nil`
- `print Int.toString 123` means `(print Int.toString) 123 (type error)`. We must write `print (Int.toString 123)`

Built-in higher order functions in ML

ML built-in functions

- In ML, built-in functions are curried, i.e., they expect their arguments as a sequence of objects separated by spaces and NOT as a tuple.
- Examples
  - `map`
  - `foldr`
  - `foldl`

`map` function

- The `map` function accepts two parameters: a function and a list of objects.
- It applies the given function to each object in the list.
- Example:

```
> map (fn x => x + 2) [1,2,3];
val it = [3, 4, 5]: int list
```

`map` definition

```
> fun map F =
    let
        fun M nil = nil
        | M(x::xs) = F x :: M xs
    in
        M
    end;
val map = fn: ('a -> 'b) -> 'a list -> 'b list

> fun square (x:real) = x*x;
val square = fn: real -> real
> val squareList = map square;
val squareList = fn: real list -> real list
> squareList [1.0,2.0,3.0];
val it = [1.0, 4.0, 9.0]: real list
```

Folding lists: `foldr` and `foldl`

- Similar to the `map` function, but instead of producing a list of values they only produce a single output value.
  - The `foldr` function folds a list of values into a single value starting from the rightmost element

  `> foldr f c [x1, …, xn]` means $f(x1, f(x2, \ldots\ f(xn, c)\ \ldots)$

  it starts at the rightmost xn with the initial value c

  ```
  > foldr (fn (a,b) => a+b) 2 [1,2,3]
  val it = 8: int
  ```

  f(1, f(2, f(3,2))) =
  1 + (2 + (3 + 2)))

  - The `foldl` function folds a list of values into a single value starting from the leftmost element

  `> foldl f c [x1, …, xn]` means $f(xn, f(xn-1, \ldots\ f(x1, c)\ \ldots)$

  it starts at the leftmost $x1$ with the initial value c

  ```
  > foldl (fn (a,b) => a+b) 2 [1,2,3]
  val it = 8: int
  ```

  f(3, f(2, f(1,2))) =
  3 + (2 + (1 + 2)))

Folding lists

- Given a list $L = [a_1, \ldots , an]$, we associate a function F with a_i (F_{a_i}), and compose all these functions by taking into account a value c as starting point
  - If the function is the product, we multiply all the elements of the list
  - If the function is adding 1, and we start with 0, we get the length of the list
- The key step is going from a_i to F_{a_i}

Definition of `foldr`

```
> fun foldr F y nil = y
    | foldr F y (x::xs) = F (x,foldr F y xs);
val foldr = fn: ('a * 'b -> 'b) -> 'b -> 'a list -> ‘b

> fun F (x,a) = x*a;
val F = fn: int * int -> int
> foldr F 1 [2,3,4];
val it = 24: int

> fun F (x,a) = a+1;
val F = fn: 'a * int -> int
> foldr F 0 [1,2,3,4];
val it = 4: int
```

F(2, F(3, F(4,1))) =
2 * (3 * (4 * 1)))

F(1, F(2, F(3, F(4,1))) =
((1 + 0) + 1) + 1) + 1)

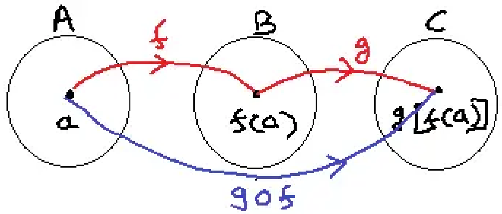
An alternative syntax

- To multiply elements of a list

```
> val prod = foldr (op *) 1;
val prod = fn: int list -> int
> prod [2,3,4];
val it = 24: int
```

We multiply the elements in the list starting from the last one multiplied by the constant 1

*(2, *(3, *(4,1))) =
2 * (3 * (4 * 1)))



Function composition in ML

Function composition

- Composition of F and G is the function H such that $H(x) = G(F(x))$
- Example:
  - $F(x) = x + 3$ and $G(y) = y^2 + 2y$,
  - $H(x) = G\big(F(x)\big) = x^2 + 6x + 9 + 2x + 6 = x^2 + 8x + 15$

In ML

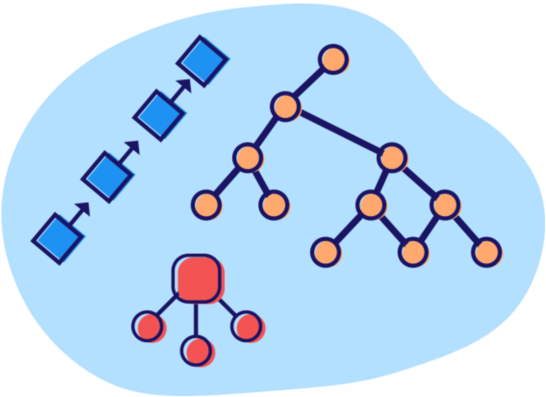
```
> fun comp (F,G,x) = G(F(x));
val comp = fn: ('a -> 'b) * ('b -> 'c) * 'a ->
'c
> comp (fn x=> x+3, fn y=>y*y+2*y, 10);
val it = 195: int
```

The operator o

```
> fun F x = x+3;
val F = fn: int -> int

> fun G y = y*y + 2*y;
val G = fn: int -> int

> val H = G o F;
val H = fn: int -> int

> H 10;
val it = 195: int
```

With the operator o we first have the most external function and then the most internal one

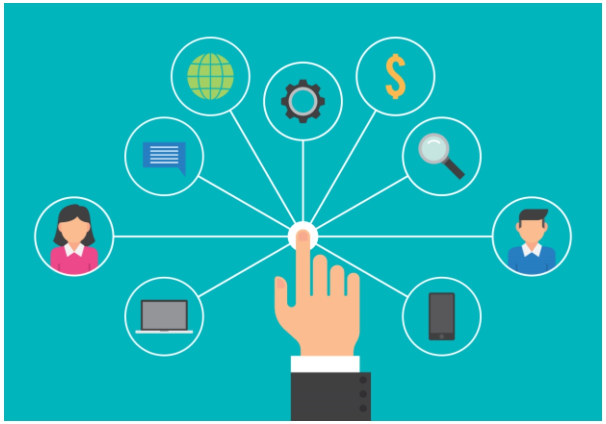
```
> op o;
val it = fn: ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```



Data types

Data types

- Data type is a high-level concept: it allows for abstracting from pure bits
- Data types: specify the values (of sequences of bits) and operations allowed on those values
  - `integer` consists of values `[-maxint .. maxint]` and operations `{+, -, *, div, mod}`
  - These operations are the only way to manipulate integers
  - Each value is wrapped in an encapsulation (its type)



User-defined types in ML

ML types

- So far
  - `int, real, string, char, bool, unit, exn, instream, outstream`
  - `T1 * T2 * ... * Tn`
  - `T1->T2`
  - `T1 list`
  - `T1 option`
- We can also
  - Rename types
  - Define new types

Type

(Parameterized) type renaming in ML

Abbreviations

- Keyword `type`

```
> type signal = int list;
type signal = int list

> val v = [1,2]: signal;
val v = [1, 2]: signal
```

- This is just an abbreviation. If we write

```
> val w = [1,2];
val w = [1, 2]: int list
```

we can then test

```
> v=w;
val it = true: bool
```

Parametrized type definitions

- In ML we can also parameterize a type definition
- Given two types `'a` and `'b` we declare `mapping` to be a type of lists of pairs of these two types

```
> type ('c,'d) mapping = ('c * 'd) list;
type ('a, 'b) mapping = ('a * 'b) list
```

Note that the type variable names are unimportant

- Example of use of this type

```
> val words = [("in",6),("a",1)] : (string,int) mapping;
val words = [("in", 6), ("a", 1)]: (string, int) mapping
```

How would you define the type?

- Give a type definition for a set of sets, where the type of elements is unspecified, and sets are represented by lists

```
> type 'a setOfSets = 'a list list;
type 'a setOfSets = 'a list list
```

How would you define the type?

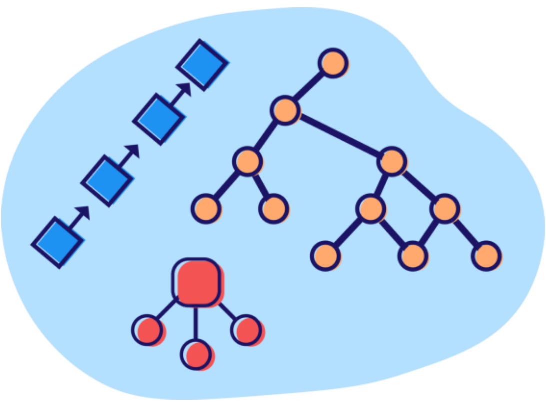
- Give a type definition for a list of triples, the first two components of which have the same type, and the third is some (possibly) different type

```
> type (’a,’b) tripleList = (’a * ’a * ’b) list;
type (’a, ’b) tripleList = (’a * ’a * ’b) list
```

What would be an example of value?

- Give an example of a value of type `(real,real) mapping`

```
> type ('c,'d) mapping = ('c * 'd) list;
type ('a, 'b) mapping = ('a * 'b) list


> val x = [(1.0,1.0),(1.0,1.1),(1.1,1.0)] : (real, real)
mapping;
val x = [(1.0, 1.0), (1.0, 1.1), (1.1, 1.0)]: (real,
real) mapping
```



Datatypes in ML

Datatypes

- Unlike type declarations, `datatype` creates new types
- Two parts
  - Type constructor, the name of the datatype
  - Data constructors, the possible values
- Example

Data constructors

```
> datatype fruit = Apple | Pear | Grape;
datatype fruit = Apple | Grape | Pear
```

Type constructor

Use of datatypes

```
> fun isApple (x) = (x = Apple);
val isApple = fn: fruit -> bool
> isApple (Pear);
val it = false: bool
> isApple(Apple);
val it = true: bool
> isApple (Cherry);
poly: : error: Value or constructor (Cherry) has not been
declared
Found near isApple (Cherry)
```

More general form of datatype definitions

- Type variables can be used to parameterize the datatype
- The data constructors can take arguments (constructor expressions)

Constructor expression

```
datatype (<parameters>) <identifier> =
      <first constructor expression> |
       ...
        <last constructor expression>
```

Constructor expressions and type variables

- Constructor expressions: data constructors that can be parameterized, e.g.,

```
Cherry of int
```

- Any expression of the form `Cherry(i)` is allowed. e.g., `Cherry (23)`
- We can also use type variables instead of `int`, e.g., `Cherry of ‘a`
- Data constructors are used to build expressions that are values for the types
- We can use these types for having types as union types, e.g.,
  - First component, type `’a`
  - Second component, if it exists, of type `’b`

Unions

- We can define a type `element` that can be a pair (‘a*’b) or a single (‘a)

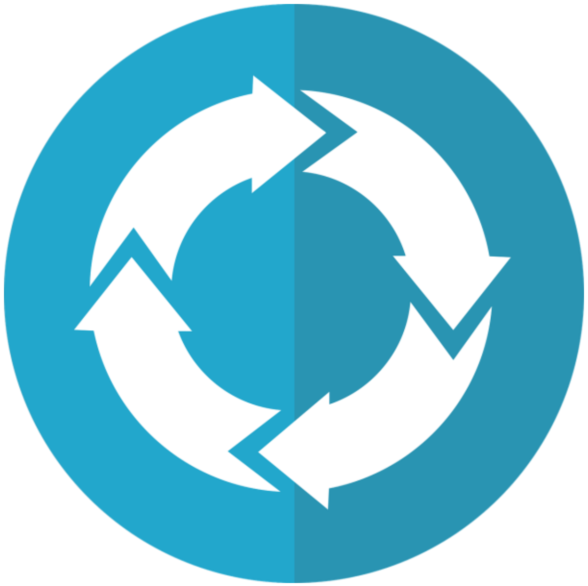
```
> datatype ('a,'b) element =
    P of 'a * 'b |
    S of 'a;
datatype ('a, 'b) element = P of 'a * 'b | S of 'a
> P ("a",1);
val it = P ("a", 1): (string, int) element
> P(1.0,2.0);
val it = P (1.0, 2.0): (real, real) element
> S(["a","b"]);
val it = S ["a", "b"]: (string list, 'a) element
```

Example

- Given a list of `(string,int) elements`, write a function `sumElList` that sums the integers in the second components, when these exist

```
> sumElList [ P("in",6), S("function"), P("as",2)];
val it = 8: int
```

```
> fun sumElList (nil) = 0
| sumElList (S(x)::L) = sumElList (L)
| sumElList (P(x,y)::L) = y + sumElList (L);
val sumElList = fn: ('a, int) element list -> int
```



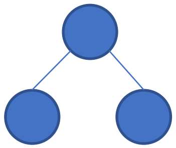
Recursively defined datatypes in ML

Recursively defined datatypes

- Binary tree:
  - Empty, or
  - Two children, each of which is, in turn, a binary tree

```
> datatype 'label btree =
    Empty |
    Node of 'label * 'label btree * 'label btree;
datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

btree



Example of data

```
> Val myTree = Node ("ML",
        Node ("as",
                Node ("a",Empty,Empty),
                Node ("in",Empty,Empty)
        ),
        Node ("types",Empty,Empty)
);

val myTree =
        Node
                ("ML", Node ("as", Node ("a", Empty, Empty), Node
("in", Empty, Empty)),
        Node ("types", Empty, Empty)): string btree
```

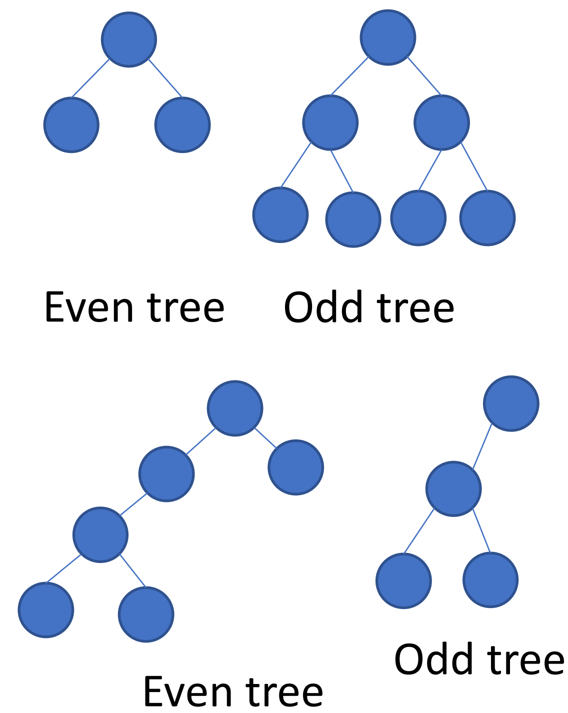
Example of function on the data

- Write a function `printTree` that, given a `string btree` prints the labels of the tree: first the label of the node, then the one(s) of the left subtree and then the one(s) of the right subtree (preorder traversal).

```
> fun printTree Empty = ()
  |printTree (Node(a,lt,rt)) = (print(a);print("
");printTree(lt);printTree(rt));
val printTree = fn: string btree -> unit

> printTree(myTree);
ML as a in types val it = (): unit
```

Mutually recursive datatypes

- Keyword `and` as with functions
- Example: Even binary trees
  - Even tree: each path from the root to a node with one or two empty subtrees has an even number of nodes
  - Odd tree: each path from the root to a node with one or two empty subtrees has an odd number of nodes

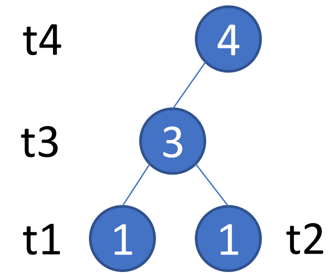


- Simple way to define it:
  - Basis: the empty tree is an even tree
  - Induction: a node with a label and two subtrees that are odd trees is the root of an even tree

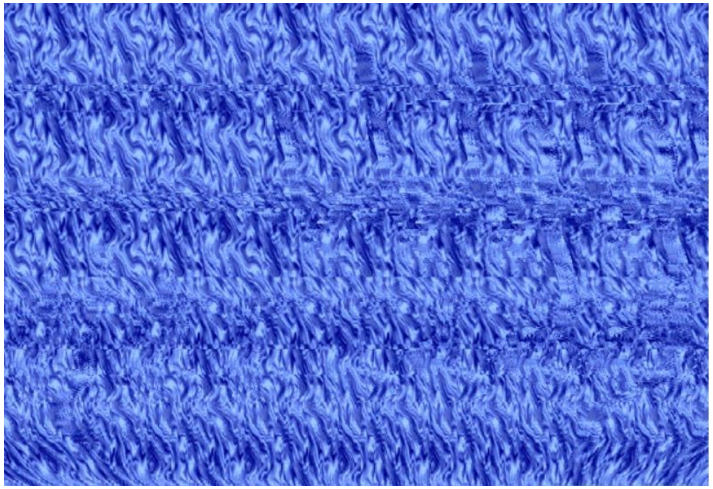
Example

```
> datatype
        'label evenTree = Empty
                | Enode of 'label * 'label oddTree * 'label oddTree
        and
        'label oddTree =
                Onode of 'label * 'label evenTree * 'label evenTree;
datatype 'a evenTree = Empty | Enode of 'a * 'a oddTree * 'a oddTree
datatype 'a oddTree = Onode of 'a * 'a evenTree * 'a evenTree
```

Example




```
> val t1 = Onode (1,Empty,Empty);
val t1 = Onode (1, Empty, Empty): int oddTree
> val t2 = Onode (1,Empty,Empty);
val t2 = Onode (1, Empty, Empty): int oddTree
> val t3 = Enode (3,t1,t2);
val t3 = Enode (3, Onode (1, Empty, Empty), Onode (1, Empty,
       Empty)): int evenTree
> val t4 = Onode (4,t3,Empty);
val t4 =
       Onode
       (4, Enode (3, Onode (1, Empty, Empty), Onode (1, Empty,
Empty)), Empty): int oddTree
```



Data abstraction

This Photo by Unknown Author is licensed under CC BY-NC-ND

Defining new data types

- When defining new data types, a user can only use existing capsules and a new type does not allow the user to define types at the same level of abstraction of the predefined types
  - It is possible to define new values
  - But the internal structure and operations are still accessible to the programmer

```
type Int_Stack = struct{
    int P[100]; // the stack proper
    int top; // first readable element
}
Int_Stack create_stack(){
    Int_Stack s = new Int_Stack();
    s.top = 0;
    return s;
}
Int_Stack push(Int_Stack s, int k){
    if (s.top == 100) error;
    s.P[s.top] = k;
    s.top = s.top + 1;
    return s;
}
int top(Int_Stack s){
    return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
    if (s.top == 0) error;
    s.top = s.top - 1;
    return s;
}
bool empty(Int_Stack s){
    return (s.top == 0);
}
```

An example

Even in case of equivalence by name, we can access the stack in its representation as an array

```
int second_from_top()(Int_Stack c){
    return c.P[s.top - 1];
    }
```

We would need … linguistic support for abstraction

- Abstraction of control
  - Hide the implementation of procedure bodies
- Data abstraction
  - Hide decisions about the representation of the data structures and the implementation of the operations
  - Example: a stack implemented via
    - A vector
    - A linked list

Abstract Data Types

- One of the major contributions of the 1970s
- Basic idea: separate the interface from the implementation
  - Interface: types and operations that are accessible to the user
  - Implementation: internal data structures and operations acting on the data types
  - Example
    - Sets have operations as `empty`, `union`, `insert`, `is_member?`
    - Sets can be implemented as vectors, lists etc.

Abstract Data Types characteristics

1. A name for the type
2. An implementation or representation for the type (concrete type)
3. Names denoting the operations for manipulating the values of the type with their types
4. For every operation, an implementation that uses the concrete type representation
5. A security capsule which separates the name of the type and those of the operations from their implementations

Concrete languages

- Different languages have different levels of support for ADT
- C:
  - Header file (.h) containing the interface/signature
  - Implementation in separate .c files
- Java, C++:
  - Object-orientation – through classes
    - Methods implementing the interface are public
    - Internal representation private
- ML:
  - Signatures and structures

Summary

- Higher order functions
- Curried functions in ML
- Built-in higher order functions in ML
- Function composition in ML
- Data abstraction
- User-defined types in ML

Readings

- Chapter 9 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Signatures, structures and functors in ML
- Intro to lambda calculus