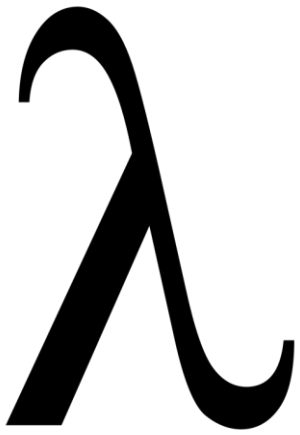


ML

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Next weeks

- May 14, 2024
  - a seminar by a colleague working with functional programming languages
- May 16, 2024 – ML Challenge
  - Teams of 3 students or for students who cannot physically attend teams of one student
  - Program in ML
  - I will send you a form for registering the group next week
  - An evaluation committee will evaluate your work
- One of the last lab classes -> exam simulation

Extra lecture

- Monday May 20 11:30 – 13:30 Aula PC B106 (lab reservation still to be confirmed)

Intermediate feedback form

- Please, fill in the form at

https://forms.gle/i7mH13wRDv61etyN6

- It will remain open until May 6

Today

- Recap
- Higher-order functions
- Curried functions
- Built-in higher-order functions
- Function composition

Recap

valOf

- How to transform a type ‘a option into the corresponding type ‘a?
  - `valOf opt`: returns the value if `opt` is `SOME`, otherwise it raises the `Option` exception

```
> valOf;
val it = fn: 'a option -> 'a

> fun convert (a: 'a option) = valOf(a);
val convert = fn: 'a option -> 'a
```

Tuple type

- Why cannot we write the following?

```
> fun f (x) = #1(x);
poly: : error: Can't find a fixed record type. Found near #1
Static Errors
```

- As the tuple could be of any arity – there is no polymorphic idea of a tuple of arbitrary arity.
- In these cases we need to use `let` so that we specify the arity of the tuple

```
> fun f(x) = let
                  val (x1,x2)=x
               in
                     x1
               end;
val f = fn: 'a * 'b -> 'a
```

Polymorphic functions

- Polymorphism: function capability to allow multiple types (“poly”=“many” + ”morph”=“form”)
- Remember: ML is strongly typed at compile time, so it must be possible to determine the type of any program without running it
- Although we must be able to identify the types, we can define functions whose types are partially or completely flexible. ML provides the type ‘a
- Polymorphic functions: functions that permit multiple types

Operators and polymorphism

Operators restricting polymorphism

- Arithmetic operators: `+`, `-`, `*` and `~`
- Division-related operators: `/`, `div` and `mod`
- Inequality comparison operators
- Boolean connectives: `andalso`, `orelse` and `not`
- String concatenation operators
- Type conversion operators, ie., `ord`, `chr`, `real`, `str`, `floor`, `ceiling`, `round` and `truncate`

Operators allowing polymorphism

- Tuple operators: `(..,..)`, `#1`, `#2`, …
- List operators: `::`, `@`, `hd`, `tl`, `nil`, `[]`
- The equality operators: `=`, `<>`

Equality types

- Types that allow the use of equality tests (= and <>)
- Integers, booleans, characters, strings, but not reals
- Tuples or lists but not functions
- Equality type is indicated with a double quote ''a
- Examples

```
> val x = (1,2);
val x = (1, 2): int * int
> val y = (2,3);
val y = (2, 3): int * int
> x=y;
val it = false: bool
> x=(1,2);
val it = true: bool
```

```
> val L = [1,2,3];
val L = [1, 2, 3]: int list
> val M = [2,3];
val M = [2, 3]: int list
> L<>M;
val it = true: bool
> L = 1::M;
val it = true: bool
```

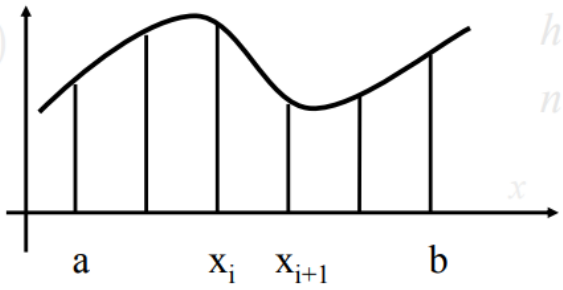
```
> identity = identity;
poly: : error: Type error in function application.
Function: = : ''a * ''a -> bool
Argument: (identity, identity) : ('a -> 'a) * ('b -> 'b)
Reason: Can't unify ''a to 'a -> 'a (Requires equality type)
```

Higher-order functions

- Functions that take functions as arguments
- Last time we saw the `trap` function

$$\int_a^b f(x)dx \approx \frac{b-a}{n} \cdot \sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2} = \sum_{i=1}^{n} \frac{b-a}{n} \cdot \frac{f(x_{i-1}) + f(x_i)}{2}$$

```
> fun trap (a,b,n,F) =
    if n<=0 orelse b-a<=0.0 then 0.0
    else let
        val delta = (b-a)/real(n)
    in
        delta * (F(a)+F(a+delta))/2.0 +
            trap (a+delta,b,n-1,F)
    end;
```



Some other higher-order functions

- We can write a function `simpleMap` that
  - takes a function F and a list $[a_1, \ldots, an]$ and produces the list $[F(a_1), \ldots, F(an)]$
  - we call it `simpleMap` to distinguish it from the built-in function `map`

simpleMap

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

> simpleMap (square, [1.0,2.0,3.0]);
val it = [1.0, 4.0, 9.0]: real list
```

Further examples

- Using a unary operator

```
> simpleMap (~, [1,2,3]);
val it = [~1, ~2, ~3]: int list
```

- Using an anonymous function

```
> simpleMap ( fn x => x*x, [1.0,2.0,3.0]);
val it = [1.0, 4.0, 9.0]: real list
```

The `reduce` function

- Defined as follows
  - List $[a_1]$ returns a_1
  - List $[a_1, \ldots, a_n]$. Reduce the tail with F obtaining b and then compute $F(a_1, b)$, e.g.,

    `reduce` ($[a_1, \ldots, a_n]$, F) = F(a_1, F(a_2, … F(a_{n_1}, a_n)))

The `reduce` function

- This means that:
  - reducing a list with the addition function returns the sum of the elements of the list
  - reducing a list with the multiplication function returns the product of the elements of the list
  - reducing a list with the logical AND returns true if all the elements of a boolean list are true
  - reducing a list with max returns the largest of the elements in the list

Definition of `reduce`

```
> exception EmptyList;
exception EmptyList

> fun reduce (F,nil) = raise EmptyList
    | reduce (F,[a]) = a
    | reduce (F,x::xs) = F(x, reduce(F,xs));
val reduce = fn: ('a * 'a -> 'a) * 'a list -> 'a
```

Infix operators: op

- In order to apply `reduce` we have to declare a function called `plus`, since "+" is infix

```
> reduce (+, [1,2,3]);
poly: : warning: (+) has infix status but was
not preceded by op.
```

- If we use `op`, we can convert an infix operator to a prefix one

```
> reduce (op +, [1,2,3]);
val it = 6: int
```

Using `reduce` to compute variance

- The variance of a list of reals $[a_1, \ldots, a_n]$ is defined as

$$\frac{\left(\sum_{i=1}^{n} {a_i}^2\right)}{n} - \left(\frac{\left(\sum_{i=1}^{n} a_i\right)}{n}\right)^2$$

- In ML

```
> fun square (x:real) = x*x;
val square = fn: real -> real
> fun plus (x:real,y) = x+y;
val plus = fn: real * real -> real
```

The `variance` function

$$\frac{(\sum_{i=1}^{n} {a_i}^2)}{n} - \left(\frac{(\sum_{i=1}^{n} a_i)}{n}\right)^2$$

- The function

```
> fun variance (L) =
    let
        val n = real(length(L))
    in
        reduce (plus,simpleMap(square,L))/n - square
        (reduce(plus,L)/n)
    end;
val variance = fn: real list -> real

> variance ([1.0,2.0,5.0,8.0]);
val it = 7.5: real
```

The `filter` function

- Select from a list those elements that satisfy a (boolean) condition

```
> fun filter (P,nil) = nil
    | filter (P,x::xs) =
        if P(x) then x::filter(P,xs)
        else filter (P,xs);
val filter = fn: ('a -> bool) * 'a list -> 'a
list

> filter (fn x => x>=10, [1,10,23,5,16]);
val it = [10, 23, 16]: int list
```

Exercise L8.1

- Use `simpleMap` to do the following:
  - Replace every negative element of a list of reals (e.g., `L = [0.0,1.0,~2.1,~2.3]`) with 0

Solution exercise L8.1

```
> val L = [0.0,1.0,~2.1,~2.3];
val L = [0.0, 1.0, ~2.1, ~2.3]: real list

simpleMap(fn(x)=>if x<0.0 then 0.0 else x, L);
val it = [0.0, 1.0, 0.0, 0.0]: real list
```

Exercise L8.2

- Use `reduce` for the following:
  - Find the maximum of a list of reals (e.g., `L = [1.1,2.2,4.4,3.3]`)

Solution exercise L8.2

```
> val L = [1.1,2.2,4.4,3.3];
val L = [1.1, 2.2, 4.4, 3.3]: real list

> reduce(fn(x,y)=> if x<y then y else x, L) ;
val it = 4.4: real
```

Exercise L8.3

- Use `filter` for the following:
  - Find the elements of a list of reals (e.g., `L = [1.1,~1.2,~1.3,1.4]`) that are greater than 0

Solution exercise L8.3

```
> val L = [1.1,~1.2,~1.3,1.4];
val L = [1.1, ~1.2, ~1.3, 1.4]: real list

> filter(fn(x)=>x>0.0, L);
val it = [1.1, 1.4]: real list
```

Exercise L8.4

- What is the effect on a list of

  `reduce(op -,L)`

Solution exercise L8.4

- What is the effect on a list of

  `reduce(op -,L)`

  $$a_1 - \left(a_2 - (a_3 - a_4)\right)$$

- Corresponding to alternating difference

  $$a_1 - a_2 + a_3 - a_4$$

```
> val L = [1,2,3,4];
val L = [1, 2, 3, 4]: int list
> reduce (op - , L);
val it = ~2: int
```

Curried functions

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)`

Exercise L8.5

- Write, in curried form, a function `applyList` that takes a list of functions and a value and applies each function to the value, producing a list of the results. If the list is empty it returns the empty list

Solution exercise L8.5

```
> fun applyList nil _ = nil
    | applyList (F::Fs) a = F(a)::(applyList Fs a);
val applyList = fn: ('a -> 'b) list -> 'a -> 'b list

> applyList [fn x=>x*2, fn x => x*x*x] 4;
val it = [8, 64]: int list
```

Exercise L8.6

- Given a function F that takes a parameter of product type with n components and the n components, define a function `curry` that applied to F produces a function `G` such that

$$G\ x_1\ \ldots\ x_n\ =\ F\ (x_1, \ldots, x_n)$$

- n should be fixed (e.g., n=3)

Solution exercise L8.6

```
> fun curry F x1 x2 x3 = F(x1,x2,x3);
val curry = fn: ('a * 'b * 'c -> 'd) -> 'a -> 'b -> 'c -> 'd

> curry (fn (x,y,z)=>x*y*z);
val it = fn: int -> int -> int -> int
> curry (fn (x,y,z)=>x*y*z) 1 2 3;
val it = 6: int
```

- We can also name `G`

```
> val G = curry (fn (x,y,z)=>x*y*z);
val G = fn: int -> int -> int -> int
> G 1 2 3;
val it = 6: int
```

Built-in higher order functions

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 $f(x1, f(x2, \dots\ f(xn, c)\ \dots)$

  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
  ```

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

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

  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
  ```

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
```

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

Exercise L8.7

- In the following exercise, use `map`, `foldr` and `foldl`
  - Turn a list of integers into a list of reals with the same values, e.g., `toReal ([1,2,3])= [1.0, 2.0, 3.0]`

Solution exercise L8.7

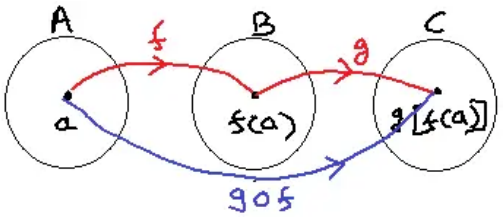
```
> val f = map real;
val f = fn: int list -> real list
> f [1,2,3];
val it = [1.0, 2.0, 3.0]: real list
```

Exercise L8.8

- In the following exercise, use `map`, `foldr` and `foldl`
  - Compute the logical AND of a list of Booleans, e.g., `andb [true, false, true] = false`

Solution exercise L8.8

```
> val andb = foldr (fn (x,y) => x andalso y)
true;
val andb = fn: bool list -> bool
> andb [true, false, false];
val it = false: bool
```



Function composition

Function composition

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

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
```

Defining a function `comp` like `o`

```
> fun comp F G =
    let
        fun C x = G(F(x))
    in
        C
    end;
val comp = fn: ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c

> fun F x = x+3;
val F = fn: int -> int
> fun G y = y*y+2*y;
val G = fn: int -> int
> val H = comp F G;
val H = fn: int -> int
> H 10;
val it = 195: int
```

Exercise L8.9

- In the following exercise, use `map`, `foldr` and `foldl`
  - Define the function `implode`, i.e., `implode[#"b",#"c"] = "bc"`

Summary

- Recap
- Curried functions
- Built-in higher-order functions
- Function composition

Next time

- User-defined types