λ

Higher order and curried functions

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

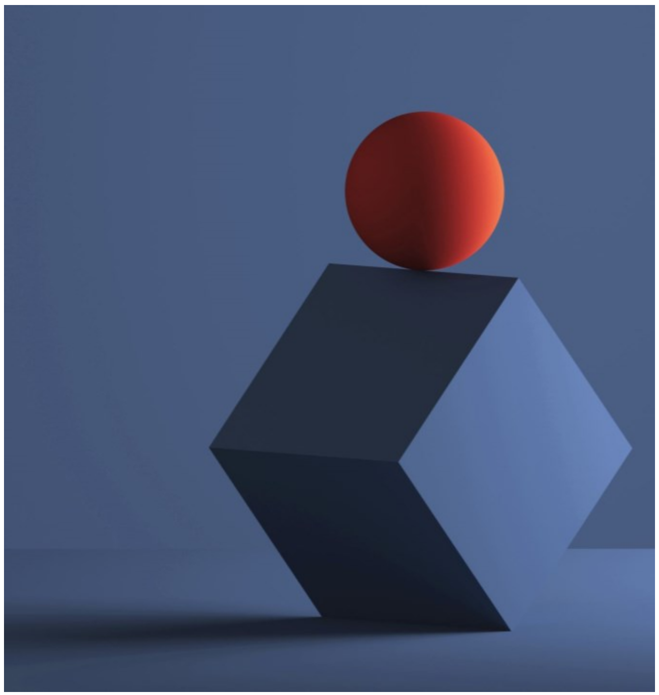
Today

- Polymorphic functions in ML
- Higher order functions in ML
- Curried functions in ML

Next lectures

- Wednesday April 9 no tutoring - ICT days
- Thursday April 10 no lecture – lab not available
- Tuesday April 15 no lecture – provette

- Next lecture is on Thursday April 17



Polymorphic functions in ML

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
- Polymorphic functions: functions that permit multiple types
- ML uses 'a for denoting generic polymorphic type

Examples

- Simple example

```
> fun identity (x) = x;
val identity = fn: 'a -> 'a
> identity (2);
val it = 2: int
> identity (2.0);
val it = 2.0: real
```

- We can even write

```
> identity (ord);
val it = fn: char -> int
```

- We can use the function twice in an expression with different types

```
> identity (2) + floor (identity (3.5));
val it = 5: int
```

Operators that restrict polymorphism

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

Operators that allow polymorphism

- Three classes in this category are:
  1. Tuple operators: `(..,..)`, `#1`, `#2`, …
  2. List operators: `::`, `@`, `hd`, `tl`, `nil`, `[]`
  3. The equality operators: `=`, `<>`

Equality types in ML

Equality types

- Types that allow the use of equality tests (= and <>)
- Integers, booleans, characters, but not reals
- Tuples or lists of equality types but not functions
- Type variables, whose values are restricted to be an equality type, are indicated with a double quote `''a`

More on equality types

- We can compare lists

```
> 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
```

- But not functions

```
> 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)
```

Examples

```
> fun identity(x) = x;
val identity = fn: 'a -> 'a
> identity(2);
val it = 2: int
> identity(2.0);
val it = 2.0: real
```

```
> fun identity_eq(x) = if (x=x)
then x else x;
val identity_eq = fn: ''a ->
         ''a
> identity_eq(2);
val it = 2: int
> identity_eq(2.0);
poly: : error: Type error in
function application.
   Function: identity_eq : ''a ->
''a
   Argument: (2.0) : real
   Reason: Can't unify ''a to real
(Requires equality type)
Found near identity_eq (2.0)
Static Errors
```

Examples

```
> fun identity(x) = x;
val identity = fn: 'a -> 'a
> identity (2);
val it = 2: int
> identity (2.0);
val it = 2.0: real
```

```
> fun identity_t(x:''a) = x;
val identity_t = fn: ''a -> ''a
> identity_t(2);
val it = 2: int
> identity_t(2.0);
poly: : error: Type error in
function application.
   Function: identity_t : ''a ->
''a
   Argument: (2.0) : real
   Reason: Can't unify ''a to real
(Requires equality type)
Found near identity_t (2.0)
Static Errors
```

Examples with lists

```
> val L: 'a list=[];
val L = []: 'a list
> 2::L;
val it = [2]: int list

> val L: 'a list=[];
val L = []: 'a list
> 2.0::L;
val it = [2.0]: real list
```

```
> val M: ''a list=[];
val M = []: ''a list
> 2::M;
val it = [2]: int list

> val M: ''a list=[];
val M = []: ''a list
 2.0::M;
poly: : error: Type error in function
application.
   Function: :: : real * real list -> real
list
   Argument: (2.0, M) : real * ''a list
   Reason: Can't unify real to ''a (Requires
equality type)
Found near 2.0 :: M
Static Errors
```

Examples with lists and functions

```
> fun first(L) = hd(L);
val first = fn: 'a list -> 'a
> first([2]);
val it = 2: int
> first([2.0]);
val it = 2.0: real
```

```
> fun first_eq(L) = if
(hd(L)=hd(L)) then hd(L) else
hd(L);
val first_eq = fn: ''a list -> ''a
> first_eq([2]);
val it = 2: int
> first_eq([2.0]);
poly: : error: Type error in
function application.
   Function: first_eq : ''a list -
> ''a
   Argument: ([2.0]) : real list
   Reason: Can't unify ''a to real
(Requires equality type)
Found near first_eq ([2.0])
Static Errors
```

Examples with lists and functions

```
> fun first(L) = hd(L);
val first = fn: 'a list -> 'a
> first([2]);
val it = 2: int
> first([2.0]);
val it = 2.0: real
```

```
> fun first_t(L:''a list) = hd(L);
val first_t = fn: ''a list -> ''a
> first_t([2]);
val it = true: bool
> first_t([2.0]);
poly: : error: Type error in
function application.
   Function: first_t : ''a list ->
''a
   Argument: ([2.0]) : real list
   Reason: Can't unify ''a to real
(Requires equality type)
Found near first_t ([2.0])
Static Errors
```

Equality types and reverse lists

- A function computing the reverse of a list function as the one below can be applied only to equality types, e.g., we cannot apply it to real values or functions

```
> fun rev1 (L) =
    if L = nil then nil
    else rev1(tl(L)) @ [hd(L)];
val rev1 = fn: ''a list -> ''a list
```

It requires equality types

The reason is the test `L=nil`

Equality types and reverse lists

```
> rev1 [1.1,2.2,3.3];
poly: : error: Type error in function application.
   Function: rev1 : ''a list -> ''a list
   Argument: [1.1, 2.2, 3.3] : ''a list
   Reason: Can't unify ''a to ''a (Requires equality type)
Found near rev1 [1.1, 2.2, 3.3]
Static Errors

> rev1 [floor,trunc, ceil];
poly: : error: Type error in function application.
Function: rev1 : ''a list -> ''a list
Argument: [floor, trunc, ceil] : (real -> int) list
Reason: Can't unify ''a to real -> int (Requires equality type)
```

Reversing lists

- We can avoid this as follows

```
> fun rev2 (nil) = nil
    | rev2(x::xs) = rev2 (xs) @ [x];
val rev2 = fn: 'a list -> 'a list
```

- We can then reverse lists of reals

```
> rev2 [1.1,2.2,3.3];
val it = [3.3, 2.2, 1.1]: real list
```

- Or even lists of functions

```
> rev2 [floor, trunc, ceil];
val it = [fn, fn, fn]: (real -> int) list
```

Testing for empty list

- An alternative way for testing if a list is empty, without forcing it to be of equality type is

```
> fun rev3 (L) =
    if null(L) then nil
    else rev3(tl(L)) @ [hd(L)];
    val rev3 = fn: 'a list -> 'a list
> rev3 [floor,trunc, ceil];
val it = [fn, fn, fn]: (real -> int) list
```

Some questions

Join this Wooclap event

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

Event code
BRAFJQ

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
```

- Of course, if you do not have further constraints, you can also specify that the formal parameter is a tuple of two items

```
> fun f(x,y) = x;
val f = fn: 'a * 'b -> 'a
```

Higher-order functions

Higher-order functions

- Some languages allow
  - passing functions as arguments to procedures
  - returning functions as results of procedures
- How do we manage the environment?

Functions as parameters

Functions as parameters

- A function is passed as a parameter to another function and then called through the actual parameter
- Call by name is a special case of functions as parameters
  - Use a function without arguments

```
{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);
 }
}
```

- Three declarations of `x`
- When `f` is called via `h`, which `x` is used?
  - Static scoping: the external `x`
  - Dynamic scoping: both of them would make sense … which one?

Binding rules

```
{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

- When a procedure is passed as a parameter, this creates a reference between a name (formal, `h`) and a procedure (actual `f`)
- Which non-local environment applies when `f` is executed, called via `h`?
    - Environment at the moment of creation of the link (deep binding): always used with static scoping
    - Environment at the moment of the call (shallow binding): Can be used with dynamic scoping

Binding policy and scope policy

- Binding policy is independent from scope policy
- Static scoping
  - Deep binding
- Dynamic scoping
  - Deep binding
  - 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);
 }
}
```

Static and deep	Dyn and deep	Dyn and shallow
x is 1	x is 4	x is 2
h(3) returns 4	h(3) returns 7	h(3) returns 5
z is 6	z is 9	z is 7

Deep vs shallow binding

- Dynamic scope
  - Possible with deep binding
    - Implementation with closure
  - Or shallow binding
    - No special implementation needed
- Static scope
  - Always uses deep binding
    - Implemented with closure

What defines the environment

- Visibility rules (based on the block structure)
- Exceptions to the visibility rules (e.g., usage of a name before its declaration)
- Scoping rules
- Rules for the parameter passing
- Binding policy

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();
}
```

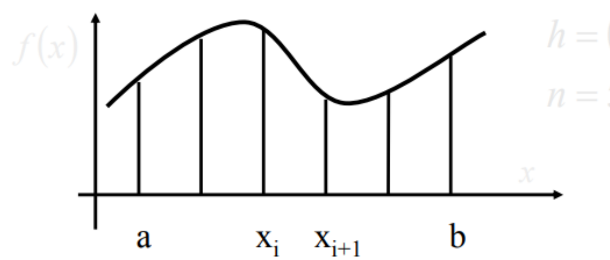
- **`void-> int`** denotes the type of the functions that take no argument and return an int
- **`void->int`** `F()` is the declaration of a function which returns a function of no argument and return value int
- **`return g`** returns the function and not its application
- gg is dynamically associated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

Higher-order functions in ML

Higher-order functions

- Functions that take functions as arguments
- Example: Approximate numerical integration $\int_a^b f(x)dx$
  - Divide the interval from a to b into n equal parts
  - Sum the areas of the n trapezoids

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



- We define a function `trap(a,b,n,F)` to do this, where the function `F` to be integrated is one of the parameters

Integration

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

```
> 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;
val trap = fn: real * real * int * (real -> real) -> real
```

Example

```
> fun square(x:real) = x*x;
val square = fn: real -> real

> trap (0.0,1.0,8,square);
val it = 0.3359375: real
```

Some other higher-order functions

- We can write a function `simpleMap` that
  - takes a function F and a list $[a_1, \dots, a_n]$ and produces the list $[F(a_1), \dots, F(a_n)]$
  - we call it `simpleMap` to distinguish it from the built-in function `map` that we will see next time
- For instance
  - `simpleMap(square,[1.0,2.0,3.0]) = [1.0,4.0,9.0]`

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

- Define a function `reduce` as follows
  - List $[a_1]$ returns a_1
  - List $[a_1, \ldots, a_n]$. Reduce the tail applying a function F that takes a pair and returns 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)))
- For instance:
  - given `fun plus (x,y) = x+y;`
  - `reduce([1,3,5], plus) = 9;`

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.
> fun plus (x,y) = x+y;
val plus = fn: int * int -> int
> reduce(plus, [1,2,3]);
val it = 6: int
```

- 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

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

- Let us define the following two functions_

```
> 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

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

- 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

- Write a function `filter` that takes as input a predicate, i.e., a boolean function and a list and selects from the list those elements that satisfy the 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
```

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)`

Summary

- Polymorphic functions in ML
- Non-generalizable types in ML
- Higher order functions in ML
- Curried functions in ML

Readings

- Chapter 7 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Memory management