λ

ML

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Today

- Recap
- Recursion
- Type inference

Recap

Variables

- Environment is modified by val-declarations (a sort of assignment) – although it is not as for imperative language variables

```
val <name> = <value>;
val <name>:<type> = <value>;
val <name> = <expression>;
```

- Example

```
> val pi = 3.14159;
val pi = 3.14159: real
> val v = 10.0/2.0;
val v = 5.0: real
```

Tuples

- Represented with rounded brackets
- Example

```
> val t = (1, 2, 3);
val t = (1, 2, 3): int * int * int
```

- We can mix types in tuple definitions

```
> val t = (4, 5.0, "six");
val t = (4, 5.0, "six"): int * real * string
```

- We can also use a complex type instead of a simple one

```
> val t = (1, (2, 3, 4));
val t = (1, (2, 3, 4)): int * (int * int * int)
```

Lists

- Represented with square brackets
- Example

```
> [1,2,3];
val it = [1, 2, 3]: int list
```

- Note that all elements of a list (unlike tuples) must be of the same type

Operators

- Tuples:
  - # i: i-th element of the tuple
- Lists
  - `hd`: head of the list
  - `tl`: tail of the list
  - `@`: concatenation of lists
  - `::`: combines an element with the list in a list
  - `explode`: explodes a string into a list of characters
  - `implode`: transforms a list of characters into a string

Functions

- In ML, just another type of value
- Represented by parametrized expressions
- Calculate a value based on parameters
  - No collateral effects
- Syntax `fn` (corresponds with λ in the λ-calculus, that we will see later)

  ```
  fn <param> => <expression>;
  ```
- Example

  ```
  fn n => n+1;
  ```
- We can directly apply the function to the parameter

  ```
  (fn n => n+1) 5;
  ```

  value 5 is associated to formal parameter n, and then the function is evaluated

Functions and names

- We can associate the functions to a name, just like values

```
> val increment = fn n => n+1;
val increment = fn: int -> int
```

- We also have a syntactic sugar notation for functions with names

```
> fun increment n = n+1;
val increment = fn: int -> int
```

- And then write

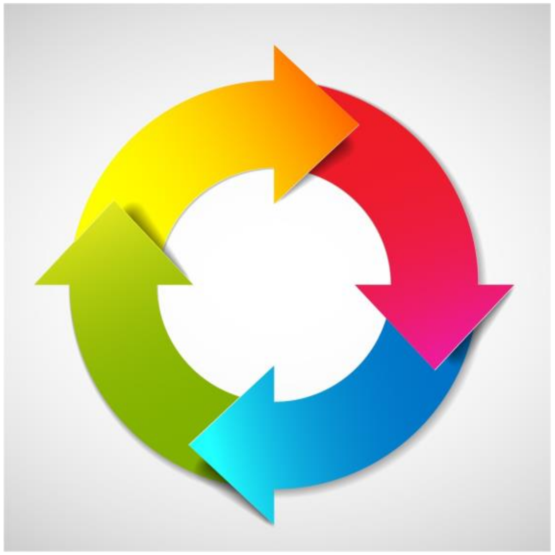
```
> increment 5;
val it = 6: int
```

Multiple arguments

- All functions in ML have exactly one parameter but this parameter can be a tuple
- Example: Largest of three reals

```
> fun max3(a:real,b,c) = (* maximum of reals *)
     if a>b then
             if a>c then a else c
     else
             if b>c then b else c;
val max3 = fn: real * real * real -> real

> max3(5.0,4.0,7.0);
val it = 7.0: real
```



Recursion

Few suggestions for thinking in a recursive way

1. Carefully read the problem to solve
2. Think about the most trivial case for our function and for which we have an immediate result (e.g., the smallest set, 0, the empty list, the empty string) – without caring about recursive call
3. Think about the recursive case:
   1. Think you are able to solve the problem in the case immediately easier
   2. Build the more complex case

Reversing a list

- Example: `reverse([1,2,3])` is `[3,2,1]`
  - Base case: empty list to empty list
  - Induction: reverse the tail of the list (recursively) and then append the head

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

> reverse [1,2,3,4];
val it = [4, 3, 2, 1]: int list
> reverse["ab","bc","cd"];
val it = ["cd", "bc", "ab"]: string list
```

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

reverse	Definition of reverse

Environment
Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

> reverse [1,2,3];
```

L	[1,2,3]
reverse	Definition of reverse

Added in call to `reverse ([1,2,3])`

Environment Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

> reverse [1,2,3];
```

L	[2,3]
L	[1,2,3]
reverse	Definition of `reverse`

Added in call to `reverse ([2,3])`

Added in call to `reverse ([1,2,3])`

Environment Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

> reverse [1,2,3];
```

`L`	`[3]`
`L`	`[2,3]`
`L`	`[1,2,3]`
`reverse`	Definition of `reverse`

Added in call to `reverse ([3])`

Added in call to `reverse ([2,3])`

Added in call to `reverse ([1,2,3])`

Environment Before the call

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

> reverse [1,2,3];
```

L	nil	Added in call to `reverse (nil)`
L	[3]	Added in call to `reverse ([3])`
L	[2,3]	Added in call to `reverse ([2,3])`
L	[1,2,3]	Added in call to `reverse ([1,2,3])`
		Environment Before the call
reverse	Definition of reverse	

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

> reverse [1,2,3];
```

L	[3]	Added in call to `reverse ([3])`
L	[2,3]	Added in call to `reverse ([2,3])`
L	[1,2,3]	Added in call to `reverse ([1,2,3])`
		Environment Before the call
reverse	Definition of `reverse`	

```
nil@[3] =[3]
```

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

> reverse [1,2,3];
```

L	[2,3]
L	[1,2,3]
reverse	Definition of `reverse`

Added in call to `reverse ([2,3])`

Added in call to `reverse ([1,2,3])`

Environment Before the call

`[3]@[2] =[3,2]`

How does the function execution works?

- The arguments are evaluated
- An addition to the environment: call-by-value

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

> reverse [1,2,3];
val it = [3, 2, 1]: int list
```

L	[1,2,3]
reverse	Definition of `reverse`

Added in call to `reverse ([2,3])`

Added in call to `reverse ([1,2,3])`

Environment Before the call

[3,2]@[1] =[3,2,1]

Nonlinear recursion

- A function can call itself recursively multiple times
- Example: Number of combinations of *k* things out of *n*
  - Written $\binom{n}{k}$
  - Can be shown to be equal to $\frac{n!}{(n-k)!k!}$
  - We can also use the following recursive definition

Combinations

- Base case. If $k = 0$ the number of ways to pick 0 out of n is 1. Similarly, if $k = n$, there is exactly one way to pick n out of n.

$$\binom{n}{0} = \binom{n}{n} = 1$$

- Induction. If $0 < k < n$ to select k out of n we can
  - reject the first thing and select k out of the remaining $n - 1$
  - pick the first thing, and pick $k - 1$ out of the remaining $n - 1$
  - formally

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

- We assume that 0 <=k<=n
- We use this to write a ML program

Combinations

```
> fun comb(n,k) = (* assumes 0 <= k <= n *)
     if k=0 orelse k=n then 1
     else comb(n-1,k) + comb(n-1,k-1);
val comb = fn: int * int -> int
```

- Without using `orelse`:

```
> fun comb (n,k) =
     if k=0 then 1
     else
          if k=n then 1 else comb(n-1,k)+comb(n-1,k-1);
val comb = fn: int * int -> int

> comb (5,0);
val it = 1: int
> comb (5,5);
val it = 1: int
> comb (5,2);
val it = 10: int
```

Mutual recursion

- Two functions can call one another recursively
- Example. A function that takes a list `L` and produces a list with the first, third, fifth etc. elements of `L`
- Two functions:
  - `take(L)` takes the first element of `L` and then alternates
  - `skip(L)` skips the first element and then calls `take`

First attempt

```
> fun take(L) =
    if L = nil then nil
    else hd(L) :: skip(tl(L));
> fun skip(L) =
    if L = nil then nil
    else take(tl(L));

> fun take(L) =
    if L = nil then nil
    else hd(L) :: skip(tl(L));
poly: : error: Value or constructor (skip) has not been
declared
Found near if L = nil then nil else hd (L) :: skip (tl (L))
```

Does not work!

Solution: keyword **and**

```
fun
          <definition of first function>
and
          <definition of second function>
and …

fun
take(L) =
          if L = nil then nil
          else hd(L) :: skip(tl(L))
and
skip(L) =
          if L = nil then nil
          else take(tl(L));
val skip = fn: ''a list -> ''a list
val take = fn: ''a list -> ''a list

> take ([1,2,3,4,5]);
val it = [1, 3, 5]: int list
```

Exercise L3.1

- Write a function that computes the factorial of n,

$$n! = 1 * 2 * \ldots * n$$

where $n \geq 1$. It does not need to work correctly for small n.

Solution L3.1

```
> fun fact(n) =
if n=1 then 1
else n * fact(n-1);
val fact = fn: int -> int

> fact 1;
val it = 1: int
> fact 10;
val it = 3628800: int
> fact 100;
Exception- Overflow raised
```

It exceeds the maxInt

Exercise L3.2

- Given an integer i and a list $L = [a_1, \dots, a_n]$, cycle L i times, i.e., produce

$$[a_{i+1}, a_{i+2}, \dots, a_n, a_1, \dots, a_i]$$

Solution L3.2

```
> fun cycle L = tl(L)@[hd(L)];

> fun cyclei (i,L) =
       if i=0 then L
       else cyclei(i-1, cycle(L));
val cyclei = fn: 'a list * int -> 'a list

> cyclei(2,[1,2,3,4]);
val it = [3, 4, 1, 2]: int list
```

Exercise L3.3

- Duplicate each element of a list, that is given the list $L = [a_1, \dots, a_n]$, produce the list $[a_1, a_1, a_2, a_2, \dots, an, a_n]$

Solution L3.3

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

> fun duplicate2(L) =
    if L=[] then []
    else [hd(L),(hd(L)]@duplicate(tl(L)));
val duplicate = fn: ''a list -> ''a list


> duplicate [1,2,3,4];
val it = [1, 1, 2, 2, 3, 3, 4, 4]: int list
```

Exercise L3.4

- Compute the length of a list.

Solution L3.4

```
> fun len(L) =
   if L=nil then 0
   else 1+len(tl(L));
val len = fn: ''a list -> int

> len [1,2,3,4];
val it = 4: int
```

Exercise L3.5

- Compute x^i where x is a real, and i a non-negative integer. It doesn't need to work for $i < 0$

Solution L3.5

```
> fun pow(x:real, i:int) = if i=0 then 1.0 else
x*pow(x,i-1);
val pow = fn: real * int -> real

> pow(2.1,3);
val it = 9.261: real
> pow(2.0,3);
val it = 8.0: real
```

Exercise L3.6

- Compute the largest (in a lexicographical sense) element of a list of strings, e.g., ([“a”,”abc”, “ab”] → “abc”). It doesn't need to work for empty lists.

Note that in the lexicographical order:

```
> "abc">"ab";
val it = true: bool
> "abc">"abb";
val it = true: bool
> "ab">"a";
val it = true: bool
```

Solution L3.6

```
> fun maxList(L: string list) =
    if tl(L)=nil then hd(L)
    else
    if hd(L)>hd(tl(L)) then maxList(hd(L)::tl(tl(L)))
    else maxList(tl(L));
val maxList = fn: string list -> string

> maxList(["a","abc","ab"]);
val it = "abc": string
```

Type inference in ML

Type inference in ML

- Types of operands and results of arithmetic expressions must agree, e.g., `(a+b)*2.0`
  - The right operand is a `real`
  - Therefore `a+b` must be a `real`
  - Therefore, so are `a` and `b`
- In a comparison (e.g., `a<=10`), both arguments have the same type, so `a` is an integer
- In a conditional, the types of the `then`, the `else` and the expression itself must all be the same

Type inference in ML

- If an expression used as an argument of a function is of a known type, the parameter must be of that type
- If the expression defining the result of a function is of a known type, the function returns that type
- If there is no way to determine the types of the arguments of an overloaded function (such as +), the type is the default (usually `integer`)

Example

```
> fun comb(n,m) = (* assumes 0 <= m <= n *)
    if m=0 orelse m=n then 1
    else comb(n-1,m) + comb(n-1,m-1);
val comb = fn: int * int -> int
```

- Result of `if` clause is an integer
- Therefore `comb(n-1,m)+comb(n-1,m-1)` is an integer
- Therefore, so are `comb(n-1,m)` and `comb(n-1,m-1)`
- Therefore `comb` returns an integer
- The expression `n-1` and `m-1` must be integers (because of -1)
- `comb` maps pairs of integers to an integer

Exercise L3.7

- In the following code

```
fun foo (a,b,c,d) =
    if a=b then c+1 else
        if a>b then c
        else b+d;
```

deduce the types of the variables and the function

Solution L3.7

- In the second line we have the expression `c+1`
- Since 1 is an integer, `c` must also be an integer
- In the third line, the expressions following the `then` and `else` must be of the same type
- Since one of these is `c`, so the type of both is integer
- So `b+d` is of integer type
- Therefore, `b` and `d` must also be integers
- Since `a` and `b` are compared on lines 2 and 3, they must be of the same type
- Therefore, `a` is also an integer
- The function type is hence

```
val foo = fn: int * int * int * int -> int
```

Exercise L3.8

- What can be inferred about the types in the following

```
fun f (a:int,b,c,d,e) =
     if a<b+c then d else e;
```

Solution L3.8

- Since `a` is compared with `b+c`, the latter sum must be an integer
- Therefore, both `b` and `c` must be integers
- We cannot tell what types `d` and `e` have, although they must both be the same
- The function type is hence

```
val f = fn: int * int * int * 'a * 'a -> 'a
```

Exercise L3.9

- What can be inferred about the types in the following

```
fun f (a:int,b,c,d,e) =
   if a<b then c else d;
```

Solution L3.9

- Since `a` is compared with `b`, these have the default type, integer
- We cannot tell what types c and d have, although they must both be the same
- The function type is hence

```
val f = fn: int * int * 'a * 'a * 'b -> 'a
```

Exercise L3.10

- What can be inferred about the types in the following

```
fun f (a:int,b,c,d,e) =
   if a<b then b+c else d+e;
```

Solution L3.10

- `b` is an integer
- `c, d` and `e` are all integers
- The function type is hence

```
val f = fn: int * int * ‘int * ‘int * ‘int -> ‘int
```

Exercise L3.11

- What can be inferred about the types in the following

```
fun f (a:int,b,c,d,e) =
   if a<b then b<c else d;
```

Solution L3.11

- `a, b` and `c` are integers
- `d` is boolean
- The function type is hence

```
val f = fn: int * int * int * bool * ’a -> bool
```

Exercise L3.12

- What can be inferred about the types in the following

```
fun f (a:int,b,c,d,e) =
   if b<c then a else c+d;
```

Solution L3.12

- `c` and `d` are integers
- `b` is also integer
- The function type is hence

```
  val f = fn: int * int * int * int * ‘a -> int
```

Exercise L3.13

- What can be inferred about the types in the following

```
fun f (a:int,b,c,d,e) =
   if b<c then d else e;
```

Solution L3.13

- `b` and `c` are integers (default)
- `d` and `e` are of the same type
- The function type is hence

```
val f = fn: int * int * int * 'a * 'a -> 'a
```

Exercise L3.14

- What can be inferred about the types in the following

```
fun f (a:int,b,c,d,e) =
   if b<c then d+e else d*e;
```

Solution L3.14

- `b` and `c` are integers (default)
- `d` and `e` are integers (default)
- The function type is hence

```
val f = fn: int * int * int * int * int -> int
```

Summary

- Recursion
- Type inference

Next time

- Local environment