λ

ML

Programmazione Funzionale
2023/2024
Università di Trento
Chiara Di Francescomarino

Today

- Recap
- Patterns
- Functions, cases and patterns

Recap

Recursion

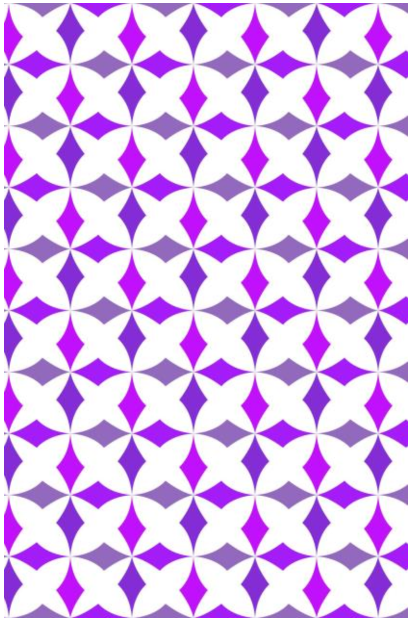
- In functional programming much more important than in imperative programming languages; used as the main mechanism for iteration

- 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
```

Type inference in ML

- Types of operands and results of arithmetic expressions must agree, e.g.,`(a+b)*2.0`
- 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
- 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`)



Patterns

Function definition: patterns

- Very powerful mechanism for defining functions
- A bit like a generalization of “case” or “switch” statements in procedural languages
- Example

  ```
  x::xs
  ```

  matches any non-empty list, with `x` set to the head and `xs` to the tail
- Function definition uses a sequence of patterns. The first that matches the argument determines the produced value

```
fun <identifier> (<first pattern>) = <first expression>
  | <identifier> (<second pattern>) = <second expression>
  ...
  | <identifier> (<last pattern>) = <last expression>;
```

Example: reverse a list

- Using patterns

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

- Without patterns

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

List of alternatives

- Note: The list of alternatives must be exhaustive, as with `if-then` clauses
  - If the list is not exhaustive, many implementations of ML only give a warning, with an error only if we actually use a parameter that does not match any of the possibilities

```
> fun reverse (nil) = nil;
poly: : warning: Matches are not exhaustive. Found near fun
reverse (nil) = nil
val reverse = fn: 'a list -> 'b list
> reverse([3]);
poly: : warning: The type of (it) contains a free type variable.
Setting it to a unique monotype.
Exception- Match raised
```

Reverse a list with patterns

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

> reverse([1,2,3])
```

We even do not try to match the second pattern

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

`as`: match pattern and assign variables

- At one time give the value to an identifier and match the value with a pattern

```
<identifier> as <pattern>
```

`as`: example

- Example: Merge two lists of integers `L` and `M`, assuming that they are be sorted (smallest first)
- Base case. If `L` is empty then the merge is `M` (and viceversa)
- Inductive case. Compare the heads of `L` and `M`. If the head of `L` is smaller add it as head and recursively call on the tail of `L` and `M`, otherwise add the head of `M` as head and recursively call on `L` and on the tail of `M`.
-
```
> fun merge (nil,M) = M
      | merge (L,nil) = L
      | merge (L as x::xs, M as y::ys) =
          if x<y then x::merge(xs,M)
          else y::merge (L,ys);
val merge = fn: int list * int list -> int list
```

Without as

- Without as it would be slightly more complicated

```
fun merge (nil,M) = M
    | merge (L,nil) = L
    | merge (x::xs,y::ys) =
        if x<y then x::merge(xs,y::ys)
        else y::merge(x::xs,ys);
```

Anonymous (or wildcard) variables

- Used when we want to match a pattern, but never need to refer to the value again

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

Multiple uses of variables in a pattern

- A variable can be used only once in a pattern
- The following is illegal

```
> fun comb (_,0) = 1
    | comb (n,n) = 1
    | comb (n,m) =
        comb(n-1,m)+comb(n-1,m-1);
poly: : error: n has already been bound in this
clause. Found near fun
comb (n, ...) = 1
```

- This should be written using `if-then` as before

Patterns allowed

- Constants, such as `nil` and `0`
- Expressions using `::`, such as `x::xs` or `x::y::xs`
- Tuples, such as `(x,y,z)`

Example

- Sum of all integers of a list of pairs of integers, e.g., given [(1,2),(3,4),(5,6)] we want to sum all the integers 1+2+3+…

```
> fun sumPairs (nil) = 0
    | sumPairs ((x,y)::zs) = x + y + sumPairs(zs);
val sumPairs = fn: (int * int) list -> int
> sumPairs [(1,2),(3,4),(5,6)];
val it = 21: int
```

Another example

- Input: list of lists of integers
- Output: Sum of these integers

```
> fun sumLists (nil) = 0
    | sumLists (nil::YS) = sumLists (YS)
    | sumLists ((x::xs)::YS) = x+sumLists (xs::YS);
val sumLists = fn: int list list -> int

> sumLists ([[1,2],nil,[3],[3,4,5]]);
val it = 18: int
```

Patterns not allowed

- Arithmetic operators, list concatenation, and real values
- Example

```
> fun length (nil) = 0
| length (xs@[x]) = 1 + length(xs);
poly: : error: @ is not a constructor Found near xs @ [x]
```

- Two more examples

```
> fun square (0) = 0
| square(x+1) = 1 + 2*x + square (x);
poly: : error: + is not a constructor Found near x + 1

> fun f(0.0) = 0
| f(x) = x;
poly: : error: Real constants not allowed in patterns
```

- But

```
> fun f(0) = 0
    | f(x) = x;
val f = fn: int -> int
```

No misspell errors

- We often use identifiers with a special meaning like `nil` (so far they are few but users can define their own with data constructors)
- We need to be careful not to misspell them – otherwise we intend a pattern that matches anything

```
> fun reverse (niil) = nil
     | reverse (x::xs) = reverse(xs) @ [x];
poly: : warning: Pattern 2 is redundant.
Found near fun reverse (niil) = nil | reverse (... :: ...) =
... ... @
[...]
val reverse = fn: 'a list -> 'a list
```

- This is not an error, but probably not what the user wanted

Exercise L4.1

- Consider the pattern

  `(x::y::zs,w)`

  Does it match the following expressions? If so, give the variable bindings

  - `(["a","b","c"],["d","e"])`
  - `(["a","b"],4.5)`
  - `([5],[6,7])`

Solution L4.1

- Consider the pattern

  `(x::y::zs,w)`

  Does it match the following expressions? If so, give the variable bindings

  - `(["a","b","c"],["d","e"])`

    Yes; `x="a"`, `y="b"`, `zs=["c"]`, and `w=["d","e"]`
  - `(["a","b"],4.5)`

    Yes; `x="a"`, `y="b"`, `zs=[]`, and `w=4.5`
  - `([5],[6,7])`

    No; the expression `y::zs` must match the empty list

Exercise L4.2

- Write the factorial function using patterns.

Solution L4.2

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


> fact 1;
val it = 1: int
> fact 10;
val it = 3628800: int
```

Exercise L4.3

- Write a function `cycle1` that cycles a list by one position using patterns. If the list is empty, return the empty list. For instance `cycle1 [1,2,3,4,5] = [2,3,4,5,1]`

Solution L4.3

```
> fun cycle1 (nil) = nil
    | cycle (x::xs) = xs @ [x];

> cycle1 [1];
val it = [1]: int list

> cycle1 [1,2,3];
val it = [2, 3, 1]: int list
```

Exercise L4.4

- Write a function `cycle_i` that cycles a list L i times using patterns. If the list is empty, return the empty list. For instance `cycle_i ([1,2,3,4,5], 3) = [4,5,1,2,3]`

Solution L4.4

```
> fun cycle_i (L,0) = L
   | cycle_i (L,i) = cycle_i (cycle(L),i-1);
val cycle_i = fn: 'a list * int -> 'a list

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

Exercise L4.5 *

- Write a function that duplicates each element of a list using patterns.

Solution L4.5

```
> fun duplicate(nil) = nil
   | duplicate(x::xs) = x::x::duplicate(xs);
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 L4.6 *

- Write a function that computes x^i using patterns.

Solution L4.6

```
> fun power (x,0) = 1
   | power (x,i) = x * power (x,i-1);
val power = fn: int * int -> int

> power (4,0);
val it = 1: int
> power (4,3);
val it = 64: int
```

Exercise L4.7

- Write a function that computes the largest of a list of reals, assuming that the list is not empty, using patterns.

Solution L4.7

```
> fun maxList([x:real]) = x
   | maxList(x::y::zs) =
      if x<y then maxList(y::zs)
      else maxList(x::zs);
poly: : warning: Matches are not exhaustive.
val maxList = fn: real list -> real


> maxList [2.0];
val it = 2.0: real
> maxList [2.0,3.1,2.7];
val it = 3.1: real
```

Exercise L4.8

- Write a function that flips alternate elements of a list using patterns. $[a_1, a_2, \ldots, a_{n-1}\ a_n]$ should become $[a_2, a_1, \ldots, a_n, a_{n-1}]$. If n is odd, leave a_n at the end.

Solution L4.8

```
> fun flip ([]) = []
   | flip ([x]) = [x]
   | flip (x::y::zs) = y::x::flip(zs);
val flip = fn: 'a list -> 'a list

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

Exercise L4.9

- Write a function that given a list L and an integer i, returns L with the i^{th} element deleted. If the length of L is less than i, return L.

Solution L4.9

```
> fun remove ([],m) = []
    | remove (x::xs,1) = xs
    | remove (x::ys,i) = x:: remove (ys,i-1);
> remove([1],5);
val it = [1]: int list
> remove([],4);
poly: : warning: The type of (it) contains a free type
variable. Setting it to a unique monotype.
val it = []: _a list
> remove([1],1);
val it = []: int list
```

Exercise L4.10 *

- Write a program to compute the square of an integer, using patterns according to the formula
$$n^2 = (n - 1)^2 + 2n - 1$$

Solution L4.10

```
> fun square(0) = 0
   | square(n) = square(n-1)+2*n-1;
val square = fn: int -> int

> square 0;
val it = 0: int
> square 6;
val it = 36: int
```

Exercise L4.11

- Write a function `flip` that takes a list of pairs of integers and orders each pair so that the smallest number is first, using patterns. For instance, `flip ([(1,2),(3,4)]) = [(2,1),(4,3)]`

Solution L4.11

```
> fun flip(nil) = nil
  | flip((x as (a:int,b))::xs) =
     if a<b then x::flip(xs) else
     (b,a)::flip(xs);
val flip = fn: (int * int) list -> (int * int)
list


> flip [(1,2),(4,3),(6,5)];
val it = [(1, 2), (3, 4), (5, 6)]: (int * int)
list
```

Exercise L4.12

- Write a function `vowel` that takes a list of characters and returns true if the first element is a vowel using patterns. For instance `vowel [#”a”,#”b”] = true`

Solution L4.12

```
> fun vowel(#"a"::ys) = true
   | vowel(#"e"::ys) = true
   | vowel(#"i"::ys) = true
   | vowel(#"o"::ys) = true
   | vowel(#"u"::ys) = true
   | vowel(_) = false;
val vowel = fn: char list -> bool

> vowel [#"a",#"b"];
val it = true: bool
> vowel [#"b",#"a"];
val it = false: bool
```

Exercise L4.13

- Let us represent sets by lists. We represent a set by a list: the elements can be in any order, but without repetitions.
- Write a function `member(x,S)` to test whether `x` is a member of set `S` using patterns

Solution L4.13

```
> fun member(_,nil) = false
   | member(x,y::ys) =
      (x=y orelse member(x,ys));
val member = fn: ''a * ''a list -> bool

> member (5,[6,7,5]);
val it = true: bool
> member (5,[6,7,8]);
val it = false: bool
```

Exercise L4.14

- Write a function that deletes an element from a set `delete(x,S)` using patterns

Solution L4.14

```
> fun delete (a,[]) = []
   | delete (b,c::ys) = if b=c then ys
      else c::delete(b,ys);
val delete = fn: ''a * ''a list -> ''a list

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

Exercise L4.15

- Write a function that inserts an element into a set `insert(x,S)` using patterns.

Solution L4.15

```
> fun insert(x,nil) = [x]
   | insert(x,S as y::ys) =
     if x=y then S else y::insert(x,ys);
val insert = fn: ''a * ''a list -> ''a list

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

Exercise L4.16

- Write a function `insertAll` that takes an element a and a list of lists `L` and inserts `a` at the front of each of these lists. For example `insertAll (1,[[2,3],[],[3]])=[[1,2,3],[1],[1,3]]`

Solution L4.16

```
> fun insertAll(a,nil) = nil
   | insertAll(a,L::Ls) =
     (a::L)::insertAll(a,Ls);
val insertAll = fn: 'a * 'a list list -> 'a list
list


> insertAll (1,[[2,3],[4,5,6],nil]);
val it = [[1, 2, 3], [1, 4, 5, 6], [1]]: int
list list
```

Exercise L4.17

- Suppose that sets are represented by lists. Write a function that takes a list, and produces the power set of the list
- If S is a set, the power set of S is the set of all subsets S' such that $S' \subseteq S$

E.g., `S=[1,2,3]`,

`powerSet(S)=[[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]`

Solution L4.17

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


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

Exercise L4.18

- Given a list of reals $[a_1, \ldots , an]$ compute

$$\prod_{i<j}(ai - aj)$$

E.g., `[1.0,2.0,3.0]`,

```
prodDiff([1.0,2.0,3.0])=(1.0-2.0)*(1.0-
3.0)*(2.0-3.0) = -2.0
```

Solution L4.18

```
> fun prodDiff1(_,nil) = 1.0
   | prodDiff1(a,b::bs) = (a-b)*prodDiff1(a,bs);
> fun prodDiff(nil) = 1.0
   | prodDiff(b::bs) =
  prodDiff1(b,bs)*prodDiff(bs);
val prodDiff = fn: real list -> real

> prodDiff [1.0,1.1,1.2,1.3,1.4];
val it = 2.88E~8: real
```

Summary

- Patterns

Next time

- Local environment