λ

ML Lab 8

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

Tutoring and mini-challenge

- Tomorrow tutoring 11:30 – 12:30 in PC A202
- ML Challenge on Thursday May 15th 10:30-12:30 during the laboratory class.
- Please register your group in the form by Tuesday May 13th (23:59) https://docs.google.com/forms/d/e/1FAIpQLSdjVknRl1y4hB3ojIUzkFcDr6TRkzR8RarMMitvfRTazIZZjQ/viewform
  - Groups can be at most composed of three students
  - For those of you who cannot attend next Thu lecture, you can participate to the mini challenge *alone* and you will have time until 23:59 of May 15th
- Please be aware that you cannot use chatgpt (or similar) to solve the mini challenge exercise.

Old exercises

Exercise 7.3

- Use `map, foldr` and `foldl` for turning a list of integers into a list of reals with the same values
- For instance
  - `toReal ([1,2,3])= [1.0, 2.0, 3.0]`

Solution 7.3

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

Exercise 7.4

- Use `map`, `foldr` and `foldl` for computing the logical AND of a list of Booleans
- For instance
  - `andb [true, false, true] = false`

Solution 7.4

```
> val andb = foldr (fn (x,y) => x andalso y)
true;

val andb = fn: bool list -> bool

> andb [true, false, false];

val it = false: bool
```

Exercise 7.5

- Use `map`, `foldr` and `foldl` for defining the function `implode`
- For instance
  - `implode[#”b”,#”c”] = “bc”`

Solution 7.5

Remember that with the composition we have first the most external function and then the most internal one, that is

```
 (g o f) x = g(f(x))
```

```
> val implode = (foldr (op ^) "") o (map str);
val implode = fn: char list -> string
> implode [#”a”,#”b”];
val it = "ab": string
```

Solution 7.5b

```
> val implode = foldr (fn (x,y) => str(x)^y) "";
val implode = fn: char list -> string
> implode [#”a”,#”b”];
val it = "ab": string
```

Exercise 7.6

- Given a binary tree datatype ’a btree

```
datatype ‘a btree =
    Empty |
    Node of ‘a * ‘a btree * ‘a btree;
```

- Write a function postOrder that returns a list of the nodes of a binary tree in postorder, where the label at the root follows the postorder traversal of the left and right subtrees (first the labels of the tree on the left, then the ones of the tree on the right and finally the root).
- For instance
  - postOrder (Node ("ML", Node ("as", Node ("a", Empty, Empty), Node ("in", Empty, Empty)), Node ("types", Empty, Empty))) = ["a", "in", "as", "types", "ML"]

Solution 7.6

```
fun postOrder (Empty) = nil
      | postOrder(Node(a,left,right)) =
      postOrder (left) @ postOrder (right) @
[a];
val postOrder = fn: ’a btree -> ’a list

> postOrder (Node ("ML", Node ("as", Node ("a",
Empty, Empty), Node ("in", Empty, Empty)), Node
("types", Empty, Empty)));
val it = ["a", "in", "as", "types", "ML"]:
string list
```

Exercise 7.7

- Given a binary tree datatype ’a btree

```
datatype ‘a btree =
    Empty |
    Node of ‘a * ‘a btree * ‘a btree;
```

- Write a function `inOrder` that returns the list of the nodes of a binary tree in inorder, where the label at the root is between the inorder traversal of the left and right subtrees, i.e., first the labels in the left tree, then the root and finally the labels in the right tree.
- For instance
  - `inOrder (Node ("ML", Node ("as", Node ("a", Empty, Empty), Node ("in", Empty, Empty)), Node ("types", Empty, Empty))) = ["a", "as", "in", "ML", "types"]`

Solution 7.7

```
fun inOrder (Empty) = nil
      | inOrder(Node(a,left,right)) =
      inOrder (left) @ [a] @ inOrder (right);
val inOrder = fn: ’a btree -> ’a list

> inOrder (Node ("ML", Node ("as", Node ("a",
Empty, Empty), Node ("in", Empty, Empty)), Node
("types", Empty, Empty)));
val it = ["a", "as", "in", "ML", "types"]:
string list
```

Exercise 7.8

- Define a type `mapTree` that is a specialization of `btree` so that it has a label type that is a set of domain-range pairs
- Define a tree `t1` that has a single node with the pair ("a",1) at the root

Solution 7.8

```
> type (’d, ’r) mapTree = (’d * ’r) btree;
type (’a, ’b) mapTree = (’a * ’b) btree
> val t1 = Node(("a",1), Empty, Empty): (string, int) mapTree;
val t1 = Node (("a", 1), Empty, Empty): (string, int) mapTree
```

Exercise 7.9

- Write a function `sumTree` for a mapTree `T` of type `(‘a,’b) mapTree` (where the order is defined by the first component). The function visits the tree and returns the sum of the second component of the label of all nodes.
- For instance
  - `sumTree (Node(("a",1), Node(("c",2), Empty, Node(("d",3), Empty, Empty)), Empty) = 6`

Solution 7.9

```
> fun sumTree Empty = 0
    | sumTree (Node((a,b),left,right)) = b + sumTree (left) +
sumTree (right);
val sumTree = fn: ('a * int) btree -> int

> val t2 = Node(("a",1), Node(("c",2), Empty, Node(("d",3),
Empty, Empty)), Empty): (string, int) mapTree;
val t2 = Node    (("a", 1), Node (("c", 2), Empty, Node (("d",
3), Empty, Empty)), Empty): (string, int) mapTree

> sumTree t1;
val it = 1: int

> sumTree t2;
val it = 6: int
```

New exercises

Exercise 8.1

- Define a signature SET with
  - Parameterized type ‘a `set`
  - Value for empty set (`emptyset`)
  - Operator to test the membership of an element to a set (`isin`)
  - Operator to add an element to a set (`addin`)
  - Operator to remove an element from a set (`removefrom`)

Solution 8.1

```
signature SET =
sig
      type ‘a set

      val emptyset: ‘a set
      val isin: “a -> “a set -> bool
      val addin: “a -> “a set -> “a set
      val removefrom: “a -> “a set -> “a set
end;
```

Note that here type is actually ”a because we have to check for equality

Exercise 8.2

- With the signature

```
signature SET =
sig
        type 'a set
end;
```

Add a definition for the structure `Set`

Solution 8.2

```
structure Set =
struct
      type ‘a set = ‘a list;
end :> SET
```

Exercise 8.3

- With the signature

```
signature SET =
sig
        type ’a set

        val emptyset: ‘a set
end;
```

Add a definition for the structure Set and test it

Solution 8.3

```
structure Set =
struct
      type ‘a set = ‘a list;

      val emptyset = [];
end :> SET
```

- Test

```
val a = Set.emptyset:
```

Exercise 8.4

- With the signature

```
signature SET =
sig
       type ‘a set

       val emptyset: ‘a set
       val isin: “a -> “a set -> bool
end;
```

Add a definition for the structure Set and test it

Solution 8.4

```
structure Set =
struct
        type ‘a set = ‘a list;

        val emptyset = [];
        fun isin _ []=false
        |isin x y::ys = (x=y) orelse isin x ys;
end :> SET
```

- Test

```
val a = Set.emptyset;
val b = Set.isin 1 a;
```

Exercise 8.5

- With the signature

```
signature SET =
sig
       type ‘a set

       val emptyset: ‘a set
       val isin: “a -> “a set -> bool
       val addin: “a -> “a set -> “a set
end;
```

Add a definition for the structure and test it

Solution 8.5

```
structure Set =
struct
       type ‘a set = ‘a list;

       val emptyset = [];
       fun isin _ []=false
       |isin x y::ys = (x=y) orelse isin x ys;
       fun addin x L = if (isin x L) then L else x::L;
end :> SET
```

- Test

```
val a = Set.emptyset;
val b = Set.isin 1 a;
val c = Set.addin 1 a;
val d = Set.isin 1 c;
```

Exercise 8.6

- With the signature

```
signature SET =
sig
       type ‘a set

       val emptyset: ‘a set
       val isin: “a -> “a set -> bool
       val addin: “a -> “a set -> “a set
       val removefrom: “a -> “a set -> “a set
end;
```

Add a definition for the structure and test it

Solution 8.6

```
structure Set =
struct
        type ‘a set = ‘a list;

        val emptyset = [];
        fun isin _ []=false
        |isin x y::ys = (x=y) orelse isin x ys;
        fun addin x L = if (isin x L) then L else x::L;
        fun removefrom _ [] = []
            |removefrom x (y::ys) = if (x=y) then ys
                                   else y::removefrom(x,ys);
end :> SET
```

- Test

```
val a = Set.emptyset;
val b = Set.isin 1 a;
val c = Set.addin 1 a;
val d = Set.isin 1 c;
val e = Set.removefrom 1 c;
val f = Set.isin 1 e;
```

Exercise 8.7

- Given the following type for trees:

```
datatype ’a T = Lf | Br of ’a * ’a T * ’a T
```

Define a signature `TREE` with the following operations besides the datatype `’a T = Lf | Br of ’a * ’a T * ’a T`

- Count the number of nodes in a tree (`countNodes`)
- Find the depth of a tree (`depth`)
- Find the mirror image of a tree (`mirror`). The mirror image of a tree is a tree in which the right and left subtrees are swapped, e.g.,
    - `mirror Br(3, Br(2,Lf,Lf), Br(5,Br(4,Lf,Lf),Lf) = Br (3, Br(4, Lf, Br(4,Lf,Lf)),Br(2,Lf,Lf))`

Solution 8.7

```
signature TREE =
       sig
       datatype ’a T = Lf | Br of ’a * ’a T * ’a T
       val countNodes : ‘a T -> int
       val depth :’a T -> int
       val mirror : ‘a T -> ‘a T
end;
```

Exercise 8.8

- Define a structure `Tree` for this signature

```
signature TREE =
       sig
       datatype ’a T = Lf | Br of ’a * ’a T * ’a T
       val countNodes : ‘a T -> int
       val depth :’a T -> int
       val mirror : ‘a T -> ‘a T
end;
```

Solution 8.8

```
structure Tree =
struct
    datatype ’a T = Lf | Br of ’a * ’a T * ‘a T
    fun countNodes Lf = 0
         |countNodes (Br(a,b,c)) = 1+countNodes(b)+countNodes(c);
    fun depth Lf = 0
         |depth (Br(a,b,c)) = if depth(b)>depth(c) then 1+depth(b)
                                else  1 + depth(c);
    fun mirror Lf = Lf
    | mirror (Br(v,t1,t2)) = Br(v,mirror(t2),mirror(t1));
end :> TREE;
```

Solution 8.8

In order to access the structure components you have to use the name of the structure. As an alternative you can type `open Tree`, however be careful opening structures – especially the predefined ones, as default functions could be overwritten.

```
> val myTree = Tree.Br(2, Tree.Br(3, Tree.Br(4, Tree.Lf, Tree.Lf),
Tree.Br(5,Tree.Lf,Tree.Lf)),Tree.Br(6, Tree.Lf, Tree.Br(7,Tree.Lf,Tree.Lf)));
val myTree =
   Br (2, Br (3, Br (4, Lf, Lf), Br (5, Lf, Lf)), Br (6, Lf, Br (7, Lf,
Lf))):
   int Tree.T
> Tree.countNodes(myTree);
val it = 6: int
> Tree.depth(myTree);
val it = 3: int
> Tree.mirror(myTree);
val it =
   Br (2, Br (6, Br (7, Lf, Lf), Lf), Br (3, Br (5, Lf, Lf), Br (4, Lf,
Lf))): int Tree.T
```

Exercise 8.9

- Given the type (‘a,’b) mapTree defined as a particular binary search tree in Exercise 7.8, such that the order is defined by the first component of the tuple:

```
type (‘a, ‘b) mapTree = (‘a * ‘b) btree;
```

- write a function lookup lt T a that searches in tree T for a pair (a, b), and, if it finds a pair (a, b), whose first component is a, it returns b
- The function lt should compare domain elements
- If there is no such a pair, return exception Missing

Solution 8.9

```
> exception Missing;
exception Missing

> fun lookup lt Empty a = raise Missing
    | lookup lt (Node((c,b),left,right)) a =
       if lt(a,c) then lookup lt left a
       else if lt(c,a) then lookup lt right a
               else b;
val lookup = fn: (’a * ’a -> bool) -> (’a * ’b) btree -> ’a ->
’b
```

Exercise L8.10

- Given the type (‘a,’b) mapTree, write a function assign lt T a b that looks in mapTree T for a pair (a, c), and, if found, replaces c by b
- If no such pair is found, assign inserts the pair (a, b) in the appropriate place in the tree

Solution 8.10

```
> fun assign lt Empty a b = Node((a, b), Empty, Empty)
        | assign lt (Node((k, v), L, R)) a b =
                if lt(a, k)
                then Node((k, v), assign lt L a b, R)
                else if lt(k, a)
                        then Node((k, v), L, assign lt R a b)
                        else Node((k, b), L, R);
val assign = fn:
(’a * ’a -> bool) -> (’a * ’b) btree -> ’a -> ’b -> (’a * ’b)
btree
```