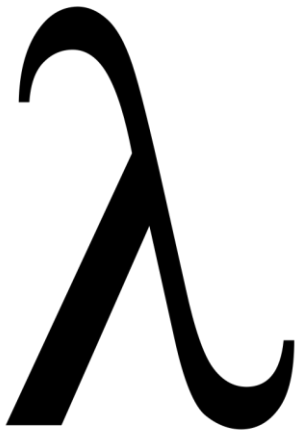


ML

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Lectures

- Today is the last lecture of the course and has been the last tutoring lab.
- In the next days I will upload some examples of theory questions
- If you have any questions, please send me an email.

Today

- Recap
- Binary Search Trees

Recap

User defined types: abbreviations

- Keyword type

```
> type signal = int list;
type signal = int list

> val v = [1,2]: signal;
val v = [1, 2]: signal
```

- This is just an abbreviation. If we write

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

we can then test

```
> v=w;
val it = true: bool
```

Parametrized type definitions

- Given two types `'a` and `'b` we declare `mapping` to be a type of lists of pairs of these two types

```
> type ('c,'d) mapping = ('c * 'd) list;
type ('a, 'b) mapping = ('a * 'b) list
```

  Note that the type variable names are unimportant

- Example of use of this type

```
> val words = [("in",6),("a",1)] : (string,int) mapping;
val words = [("in", 6), ("a", 1)]: (string, int) mapping
```

Datatypes

- Unlike type declarations, `datatype` creates new types
- Two parts
  - Type constructor, the name of the datatype
  - Data constructors, the possible values
- Example

```
> datatype fruit = Apple | Pear | Grape;
datatype fruit = Apple | Grape | Pear
```

More general form of datatype definitions

- Type variables can be used to parameterize the datatype
- The data constructors can take arguments (constructor expressions)

```
> datatype fruit = Apple | Pear | Grape | Cherry of
int;
datatype fruit = Apple | Cherry of int | Grape |
Pear
> val f = Cherry(3);
val f = Cherry 3: fruit
> val g = Apple;
val g = Apple: fruit
```

Unions

- We can define a type `element` that can be pairs (‘a*’b) or singles (‘a)

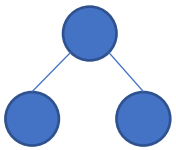
```
> datatype ('a,'b) element =
    P of 'a * 'b |
    S of 'a;
datatype ('a, 'b) element = P of 'a * 'b | S of 'a
> P ("a",1);
val it = P ("a", 1): (string, int) element
> P(1.0,2.0);
val it = P (1.0, 2.0): (real, real) element
> S(["a","b"]);
val it = S ["a", "b"]: (string list, 'a) element
```

Recursively defined datatypes

- Binary tree:
  - Empty, or
  - Two children, each of which is, in turn, a binary tree

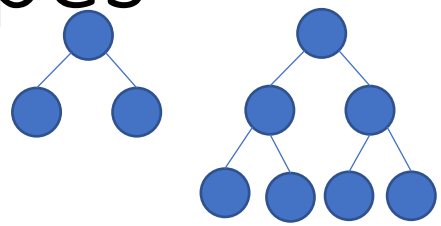
```
> datatype 'label btree =
    Empty |
    Node of 'label * 'label btree * 'label btree;
datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

btree

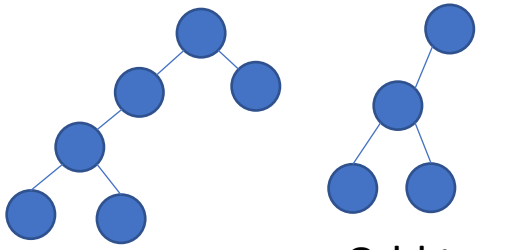


Mutually recursive datatypes

- Keyword `and` as with functions
- Example: Even binary trees
  - Even tree: each path from the root to a node with one or two empty subtrees has an even number of nodes
  - Odd tree is defined similarly



Even tree Odd tree



Even tree Odd tree

```
> datatype
          'label evenTree = Empty
                          | Enode of 'label * 'label oddTree * 'label oddTree
          and
          'label oddTree =
                          Onode of 'label * 'label evenTree * 'label evenTree;
datatype 'a evenTree = Empty | Enode of 'a * 'a oddTree * 'a oddTree
datatype 'a oddTree = Onode of 'a * 'a evenTree * 'a evenTree
```

Signatures and structures

- Structure: sequence of declarations comprising the components of the structure
  - The components of a structure are accessed using long identifiers, or paths
- Signature: similar to interface or class types

Another example

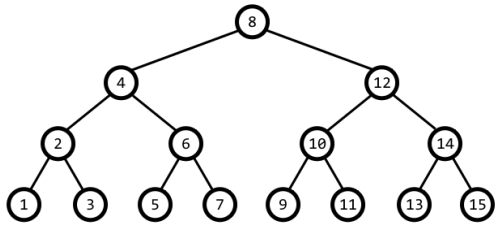
```
signature STACK =
    Sig
        eqtype 'a stack
        val empty: 'a stack
        val pop: 'a stack -> 'a option
        val push: 'a * 'a stack -> 'a stack
end;

structure Stack = struct
    type 'a stack = 'a list
    val empty = []
    val push = op::
    fun pop [] =NONE
    | pop (tos::rest) =SOME tos
end:> STACK;

val s1 = Stack.push (1, Stack.empty);
```

The declaration :> says that

- Stack is an implementation of the STACK signature
- Components not in the signature are not visible outside (including the content if the type is not specified



Binary Search Trees (BST)

Binary search trees (BST)

- Let us recall

```
> datatype 'label btree =
    Empty |
    Node of 'label * 'label btree * 'label btree;
datatype 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

- We assume an order predicate `lt(x,y)` that is
  - Transitive
  - Total
  - Irreflective

- BST property for binary labeled trees: if x is the label of a node n, then for every label y in the left subtree of n, `lt(y,x)` holds, and for every label y in the right subtree of n, `lt(x,y)` holds

Other order relations

```
fun lower (nil) = nil
   | lower (c::cs) = (Char.toLower c)::lower (cs);
val lower = fn: char list -> char list

fun strLT (x,y) =
  implode (lower (explode x)) < implode (lower (explode
y));
val strLT = fn: string * string -> bool
```

Lookup in a BST

```
datatype 'a btree = Empty | Node of 'a * 'a btree * 'a
btree

> fun lookup lt Empty x = false
    | lookup lt (Node(y,left,right)) x =
        if lt(x,y) then lookup lt left x
        else if lt(y,x) then lookup lt right x
                else true;
val lookup = fn: ('a * 'a -> bool) -> 'a btree -> 'a ->
bool
```

Example

```
> val t = Node ("ML",
        Node ("as",
                Node ("a",Empty,Empty),
                Node ("in",Empty,Empty)
                ),
        Node ("types",Empty,Empty)
        );
val t = Node ("ML", Node ("as", Node ("a", Empty, Empty), Node
("in", Empty,Empty)), Node ("types", Empty, Empty)): string
btree

> lookup strLT t "function";
val it = false: bool

> lookup strLT t "ML";
val it = true: bool
```

Insertion into BST

- Insertion does not insert into an existing tree
- It creates a new tree, with the new element added
- Recursive insert that, at each step, creates the appropriate subtree

Insertion

```
> fun insert lt Empty x = Node(x,Empty,Empty)
    |insert lt (T as Node (y,left,right)) x =
        if lt (x,y) then Node (y,(insert lt left x),right)
        else if lt (y,x) then Node (y,left,(insert lt right x))
             else T;
val insert = fn: ('a * 'a -> bool) -> 'a btree -> 'a -> 'a
btree

> insert srtLT t "function";
val it = ("ML",        Node ("as", Node ("a", Empty, Empty),
Node ("in", Node ("function", Empty, Empty), Empty)), Node
("types", Empty, Empty)): string btree
```

Deletion

- Once again, we return a modified version of the tree. This time, most of the work is in the case of equality
- We first define an auxiliary function `deletemin` which, given a BST, (i) finds the smallest element `y` in a tree `T`, and (ii) finds the tree that results after deleting this element

- Comments
  - The input to `deletemin` must be a nonempty tree
  - The smallest item will always be the left-most node, so the order relation is not needed

deletemin

```
> exception EmptyTree;
exception EmptyTree

> fun deletemin (Empty) = raise EmptyTree
        | deletemin (Node(y,Empty,right)) = (y,right)
        | deletemin (Node(w,left,right)) =
                let val (y,L) = deletemin(left)
                in (y,Node(w,L,right))
                end;
val deletemin = fn: 'a btree -> 'a * 'a btree
```

Deleting from a tree

```
> fun delete lt Empty x = Empty
 |delete lt (Node(y,left,right)) x =
        if lt (x,y) then Node(y,(delete lt left x),right)
        else if lt (y,x) then Node(y,left,(delete lt right x))
             else
                case (left,right) of
                    (Empty,r) => r |
                    (l,Empty) => l |
                    (l,r) =>
                    let val (z,r1) = deletemin(r)
                    in Node (z,l,r1)
                    end;
val delete = fn: ('a * 'a -> bool) -> 'a btree -> 'a -> 'a
btree
```

Visiting all the nodes of a tree

- Example: Sum all the values of node

```
fun sum (Empty) = 0
| sum (Node(a,left,right)) = a + sum (left) +
sum (right);
```

- Why is the type integer?

Preorder traversal

- List the label of the root
- In order from the left, list the labels of each subtree in preorder (root, followed by labesl in the left tree and then the ones in the right tree)

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

Exercise L10.1

- Write a function to list 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)

Solution exercise L10.1

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

Exercise L10.2 (*)

- Write a function to list 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.

Solution exercise L10.2

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

Exercise L10.3

- Define a type `mapTree` that is a specialization of `btree` to have 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 exercise L10.3

```
> 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 L10.4

- For this type, 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 exercise L10.4

```
> 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 L10.5

- Write a function assign `lt T a b` that looks in tree 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 exercise L10.5

```
> 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
```

Exercise L10.6

- Instantiate the function `lookup`, `insert` and `delete` to give 2-argument functions that operate on a binary search tree and a value, where the less-than function is < on reals

Solution exercise L10.6

```
> val lookup1 = lookup (op < : real*real->bool);
val lookup1 = fn: real btree -> real -> bool

> val insert1 = insert (op < : real*real->bool);
val insert1 = fn: real btree -> real -> real
btree

> val delete1 = delete (op < : real*real->bool);
val delete1 = fn: real btree -> real -> real
btree
```

Rlwrap poly

Sul terminale Linux c'è il comando "rlwrap poly", il quale vi permette di muovere il cursore liberamente all' interno della shell e di aver accesso alla funzione di history dei comandi usati recentemente all'interno della shell

Summary

- Recap
- Cases and patterns
- Signatures and structures