



BFNP – Functional Programming

Lecture 5: Finite trees

Niels Hallenberg

These slides are based on original slides by Michael R. Hansen, DTU. Thanks!!!



The original slides has been used at a course in functional programming at DTU.



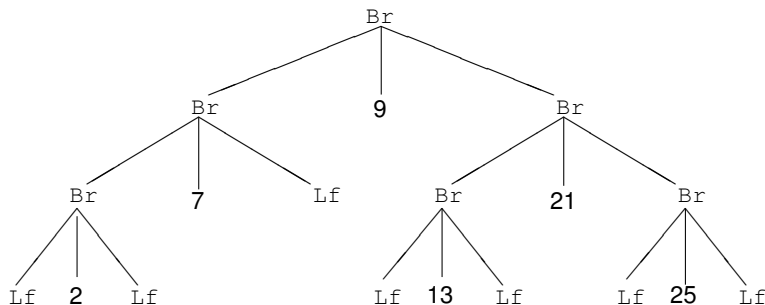
Finite Trees

- Algebraic Datatypes.
 - Non-recursive type declarations: **Disjoint union** (Lecture 4)
 - Recursive type declarations: **Finite trees**
- Recursions following the structure of trees
- Illustrative examples:
 - Search trees
 - Expression trees
 - File systems
 - . . .
- Mutual recursion, layered pattern, polymorphic type declarations



A *finite tree* is a value which may contain a subcomponent of the same type.

Example: A *binary search tree*



Condition: for every node containing the value x : every value in the left subtree is smaller than x , and every value in the right subtree is greater than x .



A *recursive datatype* is used to represent values which are trees.

```
type Tree = Lf
           | Br of Tree*int*Tree;;

Lf;;
val it : Tree = Lf

Br;;
val it : Tree * int * Tree -> Tree = <fun:clo@4>
```

The two parts in the declaration are **rules** for generating trees:

- **Lf** is a tree
- if t_1, t_2 are trees, n is an integer, then $\text{Br}(t_1, n, t_2)$ is a tree.

The tree from the previous slide is denoted by:

```
Br (Br (Br (Lf, 2, Lf) , 7, Lf) ,
     9,
     Br (Br (Lf, 13, Lf) , 21, Br (Lf, 25, Lf) ) )
```



- Recursion on the structure of trees
- Constructors `Lf` and `Br` are used in **patterns**
- The search tree condition is an **invariant** for `insert`

```
let rec insert i = function
  | Lf          -> Br(Lf,i,Lf)
  | Br(t1,j,t2) as tr ->
      match compare i j with
      | 0          -> tr
      | n when n<0 -> Br(insert i t1 , j, t2)
      | _          -> Br(t1,j, insert i t2);;
val insert : int -> Tree -> Tree
```

Example:

```
let t1 = Br(Lf, 3, Br(Lf, 5, Lf));;
let t2 = insert 4 t1;;
val t2 : Tree = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))
```



```
let rec memberOf i = function
  | Lf          -> false
  | Br(t1,j,t2) -> match compare i j with
                    | 0    -> true
                    | n when n<0 -> memberOf i t1
                    | _        -> memberOf i t2;;
val memberOf : int -> Tree -> bool
```

In-order traversal

```
let rec inOrder = function
  | Lf          -> []
  | Br(t1,j,t2) -> inOrder t1 @ [j] @ inOrder t2;;

val toList : Tree -> int list
```

gives a sorted list

```
inOrder(Br(Br(Lf,1,Lf), 3, Br(Br(Lf,4,Lf), 5, Lf))));;
val it : int list = [1; 3; 4; 5]
```



Delete **minimal element** in a search tree: `Tree -> int * Tree`

```
let rec delMin = function
| Br(Lf,i,t2) -> (i,t2)
| Br(t1,i,t2) -> let (m,t1') = delMin t1
                  (m, Br(t1',i,t2));;
```

Delete **element** in a search tree: `int -> Tree -> Tree`

```
let rec delete j = function
| Lf -> Lf
| Br(t1,i,t2) ->
    match compare i j with
    | n when n<0 -> Br(t1,i,delete j t2)
    | n when n>0 -> Br(delete j t1,i,t2)
    | _ ->
        match t2 with
        | Lf -> t1
        | _ -> let (m,t2') = delMin t2
                Br(t1,m,t2');;
```



The programs on search trees just requires an ordering on elements
– they no not need to be integers.

A polymorphic tree type is declared as follows:

```
type Tree<'a> = Lf | Br of Tree<'a> * 'a * Tree<'a>;;
```

Program texts are unchanged (though **polymorphic** now), for example

```
let rec insert i = function
    ....
    | Br(t1,j,t2) as tr -> match compare i j with
        .... ;;
val insert: 'a -> Tree<'a> -> Tree<'a> when 'a: comparison

let ti = insert 4 (Br(Lf, 3, Br(Lf, 5, Lf)));;
val ti : Tree<int> = Br (Lf,3,Br (Br (Lf,4,Lf),5,Lf))

let ts = insert "4" (Br(Lf, "3", Br(Lf, "5", Lf)));;
val ts : Tree<string>
    = Br (Lf,"3",Br (Br (Lf,"4",Lf),"5",Lf))
```




For example

```
let rec inFoldBack f t e =  
  match t with  
  | Lf          -> e  
  | Br(t1,x,t2) -> let er = inFoldBack f t2 e  
                   inFoldBack f t1 (f x er);;  
val inFoldBack: ('a -> 'b -> 'b) -> Tree<'a> -> 'b -> 'b
```

satisfies

```
inFoldBack f t e = List.foldBack f (inOrder t) e
```

It traverses the tree without building the list- For example:

```
let ta = Br(Br(Br(Lf,-3,Lf),0,Br(Lf,2,Lf)),5,Br(Lf,7,Lf));;  
  
inOrder ta;;  
val it : int list = [-3; 0; 2; 5; 7]  
  
inFoldBack (-) ta 0;;  
val it : int = 1
```



```
type Fexpr =  
  | Const of float  
  | X  
  | Add of Fexpr * Fexpr  
  | Sub of Fexpr * Fexpr  
  | Mul of Fexpr * Fexpr  
  | Div of Fexpr * Fexpr;;
```

Defines 6 **constructors**:

- Const: float -> Fexpr
- X : Fexpr
- Add: Fexpr * Fexpr -> Fexpr
- Sub: Fexpr * Fexpr -> Fexpr
- Mul: Fexpr * Fexpr -> Fexpr
- Div: Fexpr * Fexpr -> Fexpr



A classic example in functional programming:

```
let rec D = function
| Const _      -> Const 0.0
| X            -> Const 1.0
| Add(fe1,fe2) -> Add(D fe1,D fe2)
| Sub(fe1,fe2) -> Sub(D fe1,D fe2)
| Mul(fe1,fe2) -> Add(Mul(D fe1,fe2),Mul(fe1,D fe2))
| Div(fe1,fe2) -> Div(
                        Sub(Mul(D fe1,fe2),Mul(fe1,D fe2)),
                        Mul(fe2,fe2));;
```

Notice the direct correspondence with the rules of differentiation.

Can be tried out directly, as tree are "just" values, for example:

```
D(Add(Mul(Const 3.0, X), Mul(X, X)));;
val it : Fexpr =
  Add
    (Add (Mul (Const 0.0,X),Mul (Const 3.0,Const 1.0)),
      Add (Mul (Const 1.0,X),Mul (X,Const 1.0)))
```



Given a value (a float) for x , then every expression denote a float.

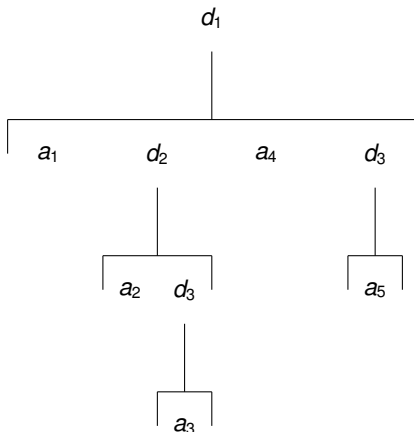
```
compute : float -> Fexpr -> float
```

```
let rec compute x = function
| Const r      -> r
| X            -> x
| Add(fe1,fe2) -> compute x fe1 + compute x fe2
| Sub(fe1,fe2) -> compute x fe1 - compute x fe2
| Mul(fe1,fe2) -> compute x fe1 * compute x fe2
| Div(fe1,fe2) -> compute x fe1 / compute x fe2;;
```

Example:

```
compute 4.0 (Mul(X, Add(Const 2.0, X)));;
val it : float = 24.0
```

Mutual recursion. Example: File system



- A **file system** is a list of **elements**
- an **element** is a file or a directory, which is a named **file system**

Mutually recursive type declarations



- are combined using **and**

```
type FileSys = Element list
and Element =
  | File of string
  | Dir of string * FileSys
```

```
let d1 =
  Dir("d1", [File "a1";
             Dir("d2", [File "a2";
                       Dir("d3", [File "a3"])]);
             File "a4";
             Dir("d3", [File "a5"])
  ])
```

The type of d1 is ?



- are combined using **and**

Example: extract the names occurring in file systems and elements.

```
let rec namesFileSys = function
  | []      -> []
  | e::es -> (namesElement e) @ (namesFileSys es)
and namesElement = function
  | File s      -> [s]
  | Dir(s,fs) -> s :: (namesFileSys fs) ;;
val namesFileSys : Element list -> string list
val namesElement : Element -> string list

namesElement d1 ;;
val it : string list = ["d1"; "a1"; "d2"; "a2";
                        "d3"; "a3"; "a4"; "d3"; "a5"]
```



Finite Trees

- concepts
- illustrative examples

Notice the strength of having trees as values.

Notice that polymorphic types and mutual recursion are NOT biased to trees.



- The Expression tree example in the book
- A simple interpreter for a statement language, to be completed in hand-in 3



To show the power of a functional programming language, we present a prototype for interpreters for a simple expression language with local declarations and a simple WHILE language.

- Concrete syntax: defined by a contextfree grammar
- Abstract syntax (parse trees): defined by algebraic datatypes
- Semantics, i.e. meaning of programs: inductively defined following the structure of the abstract syntax

succinct programs, fast prototyping

The interpreter for the simple expression language is a higher-order function:

$$eval : Program \rightarrow Environment \rightarrow Value$$

The interpreter for a simple imperative programming language is a higher-order function:

$$I : Program \rightarrow State \rightarrow State$$



Concrete syntax:

```
a * (-3 + (let x = 5 in x + a))
```

The **abstract syntax** is defined by an algebraic datatype:

```
type ExprTree = | Const of int
                | Ident of string
                | Minus of ExprTree
                | Sum    of ExprTree * ExprTree
                | Diff   of ExprTree * ExprTree
                | Prod   of ExprTree * ExprTree
                | Let    of string * ExprTree * ExprTree;;
```

Example:

```
let et =
  Prod(Ident "a",
    Sum(Minus (Const 3),
      Let("x", Const 5, Sum(Ident "x", Ident "a"))));;
```



An *environment* contains *bindings* of identifiers to values.

A `let tree` `Let (str, t1, t2)` is evaluated as in an environment *env*:

- 1 Evaluate t_1 to value v_1
- 2 Evaluate t_2 in the *env* extended with the binding of *str* to v_1 .

An evaluation function

```
eval: ExprTree -> map<string,int> -> int
```

is defined as follows:

```
let rec eval t env =  
  match t with  
  | Const n      -> n  
  | Ident s      -> Map.find s env  
  | Minus t       -> - (eval t env)  
  | Sum(t1,t2)    -> eval t1 env + eval t2 env  
  | Diff(t1,t2)   -> eval t1 env - eval t2 env  
  | Prod(t1,t2)   -> eval t1 env * eval t2 env  
  | Let(s,t1,t2)  -> let v1    = eval t1 env  
                      let env1 = Map.add s v1 env  
                      eval t2 env1;;
```



Note that the meaning of a let expression is directly represented in the program.

Example

```
let env = Map.add "a" -7 Map.empty;;  
eval et env;;  
val it : int = 35
```



An example of concrete syntax for a factorial program:

```
{Pre: x=K and x>=0}  
  y:=1 ;  
  while !(x=0)  
    do (y:= y*x;x:=x-1)  
{Post: y=K!}
```

Typical ingredients

- Arithmetical expressions
- Boolean expressions
- Statements (assignments, sequential composition, loops, ...)



- Grammar:

$aExp :: -n \mid v \mid aExp + aExp \mid aExp \cdot aExp \mid aExp - aExp \mid (aExp)$

where n is an integer and v is a variable.

- The declaration for the abstract syntax follows the grammar

```
type aExp =      (* Arithmetical expressions *)
  | N  of int      (* numbers          *)
  | V  of string   (* variables         *)
  | Add of aExp * aExp (* addition          *)
  | Mul of aExp * aExp (* multiplication    *)
  | Sub of aExp * aExp (* subtraction      *)
```

The **abstract syntax** is representation independent (no '+', '-', '(', ')', etc.), no ambiguities — one works directly on syntax trees.



- A **state** maps variables to integers

```
type state = Map<string,int>;;
```

- The meaning of an expression is a function:

```
A: aExp -> state -> int
```

defined inductively on the structure of arithmetic expressions

```
let rec A a s      =  
  match a with  
  | N n           -> n  
  | V x           -> Map.find x s  
  | Add(a1, a2) -> A a1 s + A a2 s  
  | Mul(a1, a2) -> A a1 s * A a2 s  
  | Sub(a1, a2) -> A a1 s - A a2 s;;
```




- Abstract syntax

```
type bExp =      (* Boolean expressions      *)
  | TT           (* true                    *)
  | FF           (* false                   *)
  | Eq of ....   (* equality                      *)
  | Lt of ....   (* less than                     *)
  | Neg of ....  (* negation                      *)
  | Con of ....  ;; (* conjunction                   *)
```

- Semantics $B : \text{bExp} \rightarrow \text{State} \rightarrow \text{bool}$

```
let rec B b s =
  match b with
  | TT      -> true
  | ....
```



```
type stm = (* statements *)
| Ass of string * aExp      (* assignment *)
| Skip
| Seq  of stm * stm         (* sequential composition *)
| ITE  of bExp * stm * stm  (* if-then-else *)
| While of bExp * stm;;     (* while *)
```

Example of concrete syntax:

```
y:=1 ; while not (x=0) do (y:= y*x ; x:=x-1)
```

Abstract syntax ?



An imperative program performs a sequence of state updates.

- The expression

`update y v s`

is the state that is as *s* except that *y* is mapped to *v*.

Mathematically:

$$(\text{update } y \ v \ s)(x) = \begin{cases} v & \text{if } x = y \\ s(x) & \text{if } x \neq y \end{cases}$$

- Update is a higher-order function with the declaration:

```
let update x v s = Map.add x v s;;
```

- Type?



- The meaning of statements is a function

$$I : stm \rightarrow state \rightarrow state$$

that is defined by induction on the structure of statements:

```
let rec I stm s =  
  match stm with  
  | Ass(x,a)           -> update x ( ... ) s  
  | Skip               -> ...  
  | Seq(stm1, stm2)    -> ...  
  | ITE(b,stm1,stm2)   -> ...  
  | While(b, stm)      -> ... ;;
```

Example: Factorial function



```
(* {pre: x = K and x>=0}
    y:=1 ; while !(x=0) do (y:= y*x;x:=x-1)
    {post: y = K!} *)
```

```
let fac = Seq(Ass("y", N 1),
              While(Neg(Eq(V "x", N 0)),
                    Seq(Ass("y", Mul(V "x", V "y")) ,
                        Ass("x", Sub(V "x", N 1)) )));;
```

```
(* Define an initial state *)
let s0 = Map.ofList [("x",4)];;
val s0 : Map<string,int> = map [("x", 4)]
```

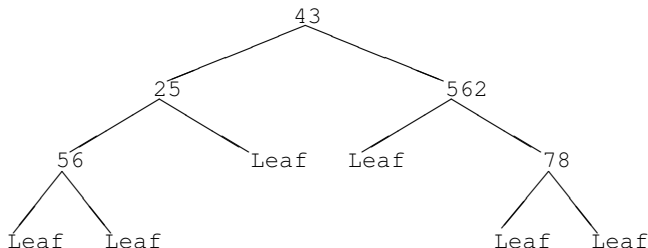
```
(* Interpret the program *)
let s1 = I fac s0;;
val s1 : Map<string,int> = map [("x", 0); ("y", 24)]
```

For the assignment 3: Simple Binary Tree



The binary tree below is both recursive and polymorphic, that is, we can put any type of data on the nodes in the tree.

```
type 'a BinTree =  
  Leaf  
  | Node of 'a * 'a BinTree * 'a BinTree  
  
let intBinTree =  
  Node(43, Node(25, Node(56, Leaf, Leaf), Leaf),  
        Node(562, Leaf, Node(78, Leaf, Leaf)))
```



For the assignment 3: Counting nodes in the tree



We will often traverse datatypes and to that is pattern matching essential.

Lets count the number of leaf-nodes and nodes.

```
let rec countNodes tree =  
  match tree with  
  | Leaf -> (1,0)  
  | Node(_,treeL, treeR) ->  
    let (ll,lr) = countNodes treeL in  
    let (rl,rr) = countNodes treeR in  
    (ll+rl,1+lr+rr);  
> countNodes intBinTree;;  
val it : int * int = (6, 5)
```

We collect the number of nodes in a pair
(numLeafNodes,numNodes). Type inference automatically finds the
type of the type variable 'a.

```
let floatBinTree =  
  Node(43.0,Node(25.0, Node(56.0,Leaf, Leaf), Leaf),  
        Node(562.0, Leaf, Node(78.0, Leaf,Leaf)))
```



There are three basic ways of traversing a binary tree:

- Pre-order traversal: node, left sub-tree, right sub-tree
- Post-order traversal: left sub-tree, right sub-tree, node
- In-order traversal: left sub-tree, node, right sub-tree

```
let rec preOrder tree =  
  match tree with  
  | Leaf -> []  
  | Node(n,treeL,treeR) ->  
    n :: preOrder treeL @ preOrder treeR;  
preOrder intBinTree;
```

We collect the elements in a list with elements of type 'a.

```
> preOrder intBinTree;;  
val it : int list = [43; 25; 56; 562; 78]
```