Functional programming, inductive data types and proofs
Third Vienna Tbilisi Summer School in Logic and Language

Pascal Manoury
Université Pierre et Marie CURIE – PARIS
Laboratoire PPS – Université Paris Diderot

24-28 September 2007

This courses will be in two parts:

- the *programming part* in which we will exercice to define programs in functional style;
- the *proving part* in which we will approch a way to ensure correctness of programs.

The connection between programs and proofs is here the correspondance between recursively defined functions and proofs by induction. There is a deep logical correspondance between proofs and programs, but this will not be the topic of this presentation. We will rather present the shallow correspondance that recursively defined functions follows naturaly inductive reasonning.

A functional programming language is oriented to the computation of *values* by mean of functional application and composition. Values may be either usual one as boolean, integers, strings, lists etc. as well as functions them selves. The theoretical bases of functional languages is the *lambda-calculus* where all is function.

A functional program is roughly a sequence of *data types and functions definitions* ended by an *expression* which is the entry point of the program to be evaluated.

Our programming language will be OCAML. It belongs to the ML's family of programming languages. It also includes imperative features and objects, but we will mainly ignore them in this course.

1 Functional programming

In OCAML the *syntax* of functional application is usually simply obtained by writting the function and its arguments: `f x y` denotes the application of function `f` to the two arguments `x` and `y`. This is the standard *prefix* notation. Parenthesis may be used. The former application may also be written `(f x y)`. Arguments may be the result of computation as in `(f (g x) (h (g y) z))`. Some usual symbols of function are written with *infix* notation, like arithmetic operators, but this is not the rule.

1.1 Basic types and operators

Any programming language provides predefined data types and basic operators for their values. They are part of the stones with which programmers build their home.

Booleans The two boolean values are writen `true` and `false`. The primitives operators are

- the negation written `not`: `(not true)` evaluates to `false`; `(not false)` evaluates to `true`.
- the conjunction written `&&` or simply `&`. It is an infix operator: `(true && x)` evaluates to the value of `x`; `(false && x)` evaluates to `false` what ever `x` may be. The conjunction operator is said to be *sequential* in the sense that it evaluates its first argument first and then, if needed, the second.
- the disjunction written `||` or `or`: `(true or x)` evaluates to `true`; `(false or x)` evaluates to the value of `x`. It is also an infix and sequential operator.

Boolean values have type `bool`. One write that the boolean constants belongs to type `bool` as `true:bool` and `false:bool`. The negation operator belongs to the set of unary functions that maps booleans to booleans. One write `not: bool -> bool`. The type of conjunction and negation is `bool -> bool -> bool`.

Numeric values Basicaly, there is two kinds of numeric values: integers and real numbers. They correspond respectively to the types `int` and `float`.

Integer values may be written as decimals: `-1073741824 ...-2 -1 0 1 2 ...1073741823`. The set of integers in OCAML is the finite interval $[-2^{31}-1, 2^{31}]$ on 32-bit processors and $[-2^{63}-1, 2^{63}]$ on 64-bit processors. All computations on integers are done *modulo* 2^{31} (or 2^{63}). Primitive operators on integers are

- unary negation and substraction, both written `-`. The value of `-n` equals that of `0-n`;
- addition and multiplication respectively written `+` and `*`. They are both infix operators;
- the integer division, written `/`. The expression `5/2` has integer value `2`.
- integer remainder, written `mod`. This is an infix operator.
- the absolute value, written `abs`.

The language also provides two special integer constants: `min_int` for the smallest value of type `int` and `max_int` for the greatest.

Real values are written as *floating point numbers*. They are written as doted decimals as `-1.5 -1.567 -1.0 0.0 1.0 1.567 1.5` and may use an exponent: `2.22e-3` for $2,22.10^{-3}$; `1.79e+3` for $1,79.10^{3}$. A zero following the dot may be omitted (as in `1.`) but a preceeding one cannot: `.1` causes a *syntax error*. Primitive operators on reals are

- the four usual arithmetic operations whose symbols are *dotted*: `+. -. *. /.`;
- absolute value and remainder are written: `abs_float abs_mod`;
- the exponentiation written `**`. It is an infix operator.
- trigonometric functions: `cos sin tan`;
- and some others we will not detail...

There are three `float` constants: `min_float`, `max_float` and `epsilon_float`. The latter being the smallest positive float `x` such that `1.0 +. x` is not equal to `1.0`. It is usefull because computations on `float` values are *approximated*. So rather the exact test `x = 0.` one will prefer `(-.epsilon_float < x) && (x < epsilon_float)`.

Division by zero Division by zero has no value for type `int` and special ones for type `float`.

Trying to evaluate `1/0` will *raise an exception* called `Division_by_zero`. When an exception is raised during an evaluation process, this process is interrupted and the all program fails unless it contains a *handler* for this exception. We will come back latter to this mechanism.

Trying to evaluate a division by `0.` will give three differents special values according to the case of the dividend. They are written `infinity` for a positive dividend, `neg_infinity` for a negative one and `nan` (*Not A Number*) for a null one. Those special values are propagated along the computation on `float` expression.

Strong typing and type inference Due to strong typing and type inference, one cannot in OCAML *overload* symbols. The integer addition has type `int -> int -> int` and can't have also type `float -> float -> float`. This is why the addition on reals is written with a plus followed by a dot (`+.`)

But one can explicitelity convert values between `int` and `float` by application of the functions `float_of_int` or `int_of_float`.

User defined functions There is no exponentiation function for type `int` but we can have 5 to the square by writting `int_of_float ((float_of_int 5) ** (float_of_int 2))` To avoid the tedious repetiton of this writting, the programmer has the ability to *define* a new function:

```
let pow n m =
 int_of_float ((float_of_int n) ** (float_of_int m))
```

The definition is introduced by the *key word* `let`. This new function is named `pow`. `n` and `m` are the *formal parameters* of the function definition. The expression following the `=` symbol is the *body* of the definition. The new function `pow` has type `int -> int -> int`. Once it is defined, 5 to the square can be obtain by evaluating the *application* `pow 5 2`.

Characters and string Characters are values of type `char`, there are numbered from 0 to 255. Characters are written as them-selves surrounded by simples quotes: `'a' ...'z' 'A' ...'Z' '0' ...'9' '?' '.' '+' '#' ...`

Strings (of characters) are values of type `string`. They are written has themselves surrounded by double quotes: `"hello Georgia!"`. The double quote inside a string must be excaped: `"hello, my name is P̈ascal"`. And so the escaping character backslash. There is an empty string written `""`. The main operators on strings is the concatenation written with the infix symbol `^`. The expression `"hello" ^ " " ^ " Georgia!"` evaluates to the `string` value `"hello Georgia!"`.

Modules and libraries A programming language provides a number of predefined data types and functions. In OCAML, they are organized with *modules* which are sets of definitions concerning a given topic. The modules `Char` and `String` contain additonal usefull functions for manipulating characters and strings. A function `f` provided by a module `M` is written as `M.f`. Remark that, in OCAML, the names of modules are required to be capitalized.

The module `String` provides the following (this is part of the OCAML documentation)

```
val length : string -> int
  Return the length (number of characters) of the given string.

val get : string -> int -> char
  String.get s n returns character number n in string s. The first
  character is character number 0. The last character is character
```

```
  number String.length s - 1. You can also write s.[n] instead of
  String.get s n.
  Raise Invalid_argument "index out of bounds" if n is outside the
  range 0 to (String.length s - 1)

val create : int -> string
  String.create n returns a fresh string of length n. The string
  initially contains arbitrary characters. Raise Invalid_argument if n
  < 0 or n > Sys.max_string_length.

val sub : string -> int -> int -> string
  String.sub s start len returns a fresh string of length len,
  containing the characters number start to start + len - 1 of string
  s. Raise Invalid_argument if start and len do not designate a valid
  substring of s; that is, if start < 0, or len < 0, or start + len >
  String.length s

val index : string -> char -> int
  String.index s c returns the position of the leftmost occurrence of
  character c in string s. Raise Not_found if c does not occur in s

val index_from : string -> int -> char -> int
  Same as String.index, but start searching at the character position
  given as second argument. String.index s c is equivalent to
  String.index_from s 0 c

[..]
```

The word `val`, which is in fact a key word of OCAML, introduces the name and type of the defined value. As OCAML is a functional language, *functions are values*. Remark the value `Sys.max_string_length` (provided by the module `Sys`) in the *specification* of `create` which indicates that strings have a maximal length.

All the libraries of the OCAML distribution are so specified. The basics are in the *core library* which corresponds to the module `Pervasives` which is the initialy *open* module. This means that you don't need to prefix values of this module with the name of the module to use them.

The *standard library* includes 39 modules corresponding to usual data structures and usefull programing utilities. 9 other libraries are also available, each one being a module, providing more specialized tools.

The `unit` type There is a particular type named `unit` which contains only one value written `()` (opening and closing parenthethis). It is used for functions whose result or argument does not matter. Typicaly, *input and output* functions make use of it.

```
val print_string : string -> unit
  Print a string on standard output.

val read_line : unit -> string
  Flush standard output, then read characters from standard input
  until a newline character is encountered. Return the string of all
  characters read, without the newline character at the end.
```

Polymorphic comparisons The infix operator `=` checks the equality between two values. This operator has a *polymorphic* type written `'a -> 'a -> bool`. The result of a comparison is a boolean value and it can be applied to values of *any type*. The symbol `'a` is used as a *type variables* that may be *instanciated* with any type expression during the type inference process. The only requirement is that the two arguments of the equality test must have the same type.

Do not confuse here polymorphism and overloading of symbols since "overloading" means that several functions are attached to a symbol while "polymorphism" means that a same function works properly for values of any type.

Other polymorphic comparison operators are

- `<>` negation of equality
- `<` "smaller than" comparison
- `>` "greater than" comparison
- `<=` "smaller or equal than" comparison
- `>=` "greater or equal than" comparison

Note that those operators will raise `Invalid_argument` if one try to apply them to *functional values*, that is to say a value whose type contains an arrow (`->`).

Using comparison: alternative Using the `<=` operator we can define the `min` and `max` functions such that

- `(min x y)` evaluates to `x` if `(x <= y)`, and to `y` otherwise
- `(max x y)` evaluates to `y` if `(x <= y)`, and to `x` otherwise

Programming languages provide this kind of alternative definition by mean of the `if-then-else` construction. Let's use it to define `min` and `max`

```
let min x y =
  if (x <= y) then x
  else y

let min x y =
  if (x <= y) then y
  else x
```

Both functions have polymorphic type `'a -> 'a -> 'a`.

The words `if`, `then` and `else` are key words of the alternative construction. They are used to *control the evaluation* flow: the condition between `if` and `then` (which must be a `bool` value) is first evaluated; and according to its value, the evaluation process is oriented to the first expression (between `then` and `else`) or to the second (after the `else`). Note that in a `if-then-else` construction *only one* of the alternative is evaluated.

Recursive functions Now we have enough to introduce the powerfull feature of functional programming: *the recursion*. Let's begin with the well known factorial function.

```
let rec fac n =
  if n=0 then 1
  else n * (fac (n-1))
```

We have `fac: int -> int`. Note that in case of recursive definitions, the key word `rec` must follow the `let`. The name of the defined function comes after (here: `fac`).

The evaluation process of the application of `fac` may be described as:

1. `(fac 3)` evaluates to `(3 * (fac 2))`
2. `(fac 2)` evaluates to `(2 * (fac 1))`
3. `(fac 1)` evaluate to `(1 * (fac 0))`
4. `(fac 0)` evaluates to `1`
5. so, `(fac 1)` evaluate to `1 * 1=1`
6. and so, `(fac 2)` evaluates to `2 * 1=2`
7. and finaly, `(fac 3)` evaluates to `3 * 2=6`

The factorial is well defined for *naturals*. What happens with negative arguments? Trying to evaluate `fac (-1)`[1] will cause the following error message:

```
Stack overflow during evaluation (looping recursion?).
```

Actually, the *"function loops"*

1. `fac (-1)` evaluates to `(-1) * (fac (-2))`
2. `fac (-2)` evaluates to `(-1) * (fac (-3))`
3. `fac (-3)` evaluates to `(-1) * (fac (-4))`
4. and so on...

The case where the argument equals `0`, the *base case of the recursion*, is never reached. In the mathematical world, this is actually the case that the base case will never be reached as integers can infinitely decrease. In the computers world the set of `int` values is finite and we said that computation on `int` values are made modulo `max_int`. So we should have (on 32-bit processors)

`fac (-1)` evaluates to `(-1) * (fac (-2))`

...

`fac -1073741823` evaluates to `(-1073741823) * (fac (-1073741824))`
which is `(-1073741823) * (fac min_int)`

`fac min_int` evaluates to `min_int * (fac max_int)` (and `max_int` is positive)

...

`fac 1` evaluate to `1 * (fac 0)`

`fac 0` evaluates to `1`

so, `fac 1` evaluate to `1`

...

[1] Remark the parenthesis around `-1`. They are required to avoid ambibuity with the substraction.

and finally, `fac (-1)` evaluates to *a very big number!*

But, what actually happens is that the processor has no room enough to reach the base case. Let's try to explain why. To evaluate an expression like `3 * (fac 2)` one needs to proceed to the evaluation of `(fac 2)` *remembering* that it will have to multiply the result by `3`. This *"remembering"* consumes *memory space*. A lot of such *"remembering"*'s consume a lot of space and it can be to much for achieving the computation. The memory consumed along a recursive process of evaluation is a *stack* where the delayed computation are *pushed* until the base case is encountered. At this time the pushed elements are *poped* one after one, from the last pushed to the first, to terminate the calculus.

When defining a recursive function attention must be paid to the reachability of the base case. This is the warranty of the *well foundness* of recursive definitions. So let's patch our `fac` definition in this way:

```
let rec fac n =
  if n < 0 then raise (Invalid_argument "fac")
  else if n = 0 then 1
  else n * (fac (n - 1))
```

Note the chained `if-then-else` construction. In the case the argument of `fac` is negative, the evaluation process will stop raising the exception `Invalid_argument`. This is done by the primitive function `raise`.

Local definition Our patched definition of `fac` have a design default, even when the argument is correct (*i.e.* positive), the evaluation process will check its non negativity at each step of the recursively looping evaluation process. We can avoid this bad feature using a *local definition*.

```
let fac n =
  let rec loop n =
    if n=0 then 1
    else n * (loop (n-1))
  in
    if n < 0 then raise (Invalid_argument "fac")
    else (loop n)
```

The local definition is between the second `let` and the key word `in`. A construction as `let x = e1 in e2` is an expression which defines the value of `x` as being the one of `e1` for the evaluation of `e2`. So `x` is said to be local to `e2`.

Note that only the inner function `loop` has to be recursive and that the formal parameter `n` of `fac` is not the formal parameter `n` of `loop`.

In this way, the negativity of the argument of `fac` is checked only one time. The function `loop` can *assume* that its argument is not negative.

Local definitions can also be used to avoid the repetition of redondants calculus. For instance, if we note that $x^{2n} = (x^n)^2$ and $x^{2n+1} = x(x^n)^2$, to compute x^n we can first compute (one time) $x^{n/2}$ and then x^n according to the parity of n. Let's see how local definitions can be used for that

```
let rec pow x n =
  match n with
    0 -> 1
  | 1 -> x
  | n ->
     let r = pow x (n/2) in
     let r2 = r * r in
```

```
        if (even n) then r2
        else x * r2
```

Assuming that even checks the even parity of its argument.

Tuples Functions return one value and it may be needed to have computations which return more than one value. For instance one may want to split a string as "hello world" into the two words it contains. For this, programming languages provide the ability of embeding values in *data structures*. In our case we can form the pair of the two words as ("hello", "world"). The type of this compound value is string * string. There are two functions to *acces* to the components of a pair

```
val fst : 'a * 'b -> 'a
  Return the first component of a pair.

val snd : 'a * 'b -> 'b
  Return the second component of a pair.
```

Note that they are polymorphic functions and that the type of the elements of a pair may differ.

Consider, for instance, the function that split a string in two worlds. More precisely, words are separeted by a space character. Our function splits the string into two substrings: its first word and the remainding of the string

```
let head_word s =
  let i = String.index s ' ' in
    (String.sub s 0 i,
     String.sub s (i+1) ((String.length s) - (i+1)))
```

This function has type string -> string * string. It fails with exception Not_found if the string contains no space character.

More generaly one can form tuples of any length as (1, "one", "ein") which has type int * string * string. But the functions fst and snd apply only one pairs (elements of the type 'a * 'b). If he needs them, the programer has to define the projections for tuples others than pairs. For triples:

```
let fst_3 (x, y, z) = x

let snd_3 (x, y, z) = y

let thd_3 (x, y, z) = z
```

In pratice, one rarely needs to explicitely defined this functions but will use the *pattern matching* mechanism we will see later.

Beware that functions defined with two arguments cannot be applyed to a pair. A pair is *one* value, even if compound of two elements. For instance, the function String.get has type string -> int -> char. So trying to evaluate String.get ("hello world", 5) will issue to the typing error:

```
This expression has type string * int but is here used with type string
```

Lists: inductive data structure A *list* is a sequence of values. In a strongly typed language as OCAML, all values of a list must have the same type. The *generic* data type of lists in OCAML is a *parametrized* type written 'a list. It is defined in terms of *constructors*

- a constant, written `[]`, for the empty list
- an operator, written `::`, for adding an element in front of an already existing list. It is an infix operator. it is called the *cons* function.

This is an *algebraic* data type. The set of values of `'a list` can be *inductively defined* as.

> For all type `'a`
>
> - `[]` belongs to `'a list`;
> - if `x` belongs to `'a` and `xs` to `'a list` then `x::xs` belongs to `'a list`.

So the list of the 3 first positive integers is `0::1::2::[]`. As lists are of frequent use, there is a syntactical shortcut for such a value: `[0;1;2]`. The type parameter of the elements is `int`, so the all value has the type `int list`. The type variable `'a` has been instanciated to `int` by the type inference mechanism.

The main operator on lists is the concatenation. It is written as the infix operator `@`. Other current functions for lists are provided by the module `List`.

Pattern matching So lists are either the empty one `[]`, either of the *form* `x::xs` for some *head* value `x` and *tail* list `xs`. Imagine we want to know if some value `v` occurs or not in a given list[2]. According to the inductive structure of lists, we can apply the following reasonning

- if the list is empty then the result is `false`
- if the list has a head `x` and a tail `xs`, that is to say, has the form `x::xs` then
  - if `x = v` then the result is `true`
  - else the result is the one of checking if `v` occurs in `xs`

So we have the theoretical way to define the function *mem* (for *mem*ber) whose value is `true` or `false` according to the occuring or not of a value in a list. This theory can be implemented in OCAML (and all ML dialects) using the powerfull construction of *pattern matching.*

```
let rec mem v xs =
  match xs with
    [] -> false
  | x::xs' -> if (x = v) then true else (mem v xs')
```

The key words `match with` enclose the expression being analyzed in terms of its pattern (its form). Then follows the sequence of intended possible patterns separated with the vertical bar `|`. For each case, the pattern itself is written on the left side of the arrow `->` (characters minus and greater-than); the value of the function associated to each pattern is written of the right side of the arrow. Moreover, the pattern expression can *name* the components of the analyzed value and those names may be used in the associated right expression.

We can also set more shortly

```
let rec mem v xs =
  match xs with
    [] -> false
  | x::xs' -> (x = v) or (mem v xs')
```

[2]This function is provided by the standard module `List`.

To summarize, a pattern matching construction like

```
match ... with
  [] -> ...
| x::xs -> ...
```

operates both

- *case analysis* of the data structure
- *binding* of data structure components to local names

Tail recursion Following the recursive function schema of `mem` we can define the function that counts the number of occurrences of a value in a list.

```
let rec count v xs =
  match xs with
    [] -> 0
  | x::xs' -> if (x = v) then 1 + (count v xs') else (count v xs')
```

In the positive alternative of the matching case `x::xs'` the evaluation process must have get the value of `(count v xs')` *before* it can add 1 to get the final result. Here, we can avoid the calculations of `1 +` to be pushed. We do it by in adding an argument to the recursive function. This extra argument is called an *accumulator*. It will contains along the recursive evaluation process the intermediate results

```
let count v xs =
  let rec loop n xs =
    match xs with
      [] -> n
    | x::xs' -> if (x = v) then (loop (n+1) xs) else (loop n xs)
  in
    loop 0 xs
```

The recursive local function `loop` is said to be *tail recursive*. We have used a local definition of the recursive exploration of the list to keep the same type for the *gobal* function `count`. Note that the searched value `v` is not an argument of `loop`. It is actually not needed because, this value is keept constant along the exploration, and due to the *scope* of variables, the value `v` is known by the inner definition.

The correctness of this tail recursive version relies on the fact that the local recursive function `loop` is first called with the wright accumulator value: `0`.

A *trace* of the evaluation process of an application of `count` is

1. `(count true [true; false; true; flase])` evaluates to `(loop [true; false; true; false] 0)`
2. `(loop [true; false; true; false] 0)` evaluates to `(loop [false; true; false] 1)`
3. `(loop [false; true; false] 1)` evaluates to `(loop [true; false] 1)`
4. `(loop [true; false] 1)` evaluates to `(loop [false] 2)`
5. `(loop [false] 2)` evaluates to `(loop [] 2)`
6. `(loop [] 2)` evaluates to 2

7. then (count true [true; false; true; flase]) evaluates to 2

The gain here is that there has been no need push pending calculus and then the number of steps of the recursive evaluation process is almost divided by 2.

Note that as the `if-then-else` construction is functional, we could have also write

```
let count v xs =
  let rec loop n xs =
    match xs with
      [] -> n
    | x::xs -> loop xs (if (x=v) then n+1 else n)
  in
    loop 0 xs
```

with better emphasis the *tail recursion schema*: the recursive application of `loop` is unique and outmost in the recursive expression.

Using and handling exceptions One wants to define the function that give the position of the first occurence (the leftmost) of a value in a list. The point is: what will be the value of the function if the value has no occurence ? In a strongly typed language, the answer is not always obvious because we have to determine a resulting value of the intended type for the erroneous argument such that we can be sure it will never be regular value. Instead of this, exceptions are a better way to signal exceptionals or unexpected situations.

So a proper way to define the index search function is

```
let rec index v xs =
  match xs with
    [] -> raise Not_found
  | x::xs' -> if (x = v) then 0 else 1 + (index v xs')
```

As an exercise: give a tail recursive version of this function.

Imagine now that we have a list of lists (a value of type `'a list list`) and we want to find the (first) position of an `'a` value in it. The position is here double: the position of the sublist in which the value occurs and the position in this sublist. The result of the function will be a pair of `int`.

The idea of the algorithm is the following:

- if there is no more list, then fail
- else
  - if the value is in the first sublist then the result is $(0, i)$ with 0 for the index of the sublist and i for the index of the value in the sublist
  - else search in the remainding list and add 1 to the index of the sublist in the obtained result.

This can be done in this naive literal way

```
let rec index2 v xss =
  match xss with
    [] -> raise Not_found
  | xs::xss' ->
```

```
    if List.mem v xs then
      (0, index v xs)
    else
      let (i1, i2) = index2 v xss' in
        (i1+1, i2)
```

Remark that one can use a tuple in a `let` construction.

This definition has the default of exploring a first time `xs` to check the occurence of `v` and a second time to compute its index. Using the fact that the function `index` raises an exception whenever `v` does not occurs in `xs` we potentialy already have the two informations. So we have either a value either the signal that we have to search again. To actualize this potentiality we need to *catch* the eventual raising of the exception in order to resume the computation. This is done in OCAML in this way

```
let rec index2 v xss =
  match xss with
    [] -> raise Not_found
  | xs::xss' ->
      try
        (0, index v xs)
      with Not_found ->
        let (i1, i2) = index2 v xss' in
          (i1+1, i2)
```

The exception handling construction is done with the two keys words `try` and `with`. Its meaning can be phrased as: `try` to compute `index` for `v` in `xs` and if it succed, we have finished; but if it fails `with Not_found` exception, we continue the computation on the rest of the list. The `try-with` construction of this second definition has replaced the `if-then-else` of the first one.

This way to use exceptions to *control* the evaluation process is related to *programming with continuations*. Actually, the exception mechanism of programming languages is an optimized particular case of the more general continuation mechanism.

Sorted lists Ordering values storage data structure as lists presents some advantages for some usual operations as searching, adding, removing values. It may spped such operation by avoiding the explorate the full length of the list.

Let's see it with the `mem` function

```
let rec mem v xs =
  match xs with
    [] -> false
  | x::xs' ->
      if (x < v) then false
      else (x = v) or (mem v xs')
```

According to the ordering relation between the searched value `v` and the head value `x`, it is possible to decide to leave the search before the end of the list is reached.

So let's study how, given any list, we can compute a list containing the same elements, but rearranged according to the ordering defined by the comparison operator `<=`.

The simplest *sorting algorithm* in functional programming is the *insertion sort*. It is based on a function which inserts a value at its place in an already sorted list. The insertion is iterated with all the elements of the list to be sorted; starting from the empty list, the resulting sorted list is thus constructed.

```
let rec insert v xs =
  match xs with
    [] -> [v]
  | x::xs' -> if (v <= x) then v::xs else x::(insert v xs')

let ins_sort xs =
  match xs with
    [] -> []
  | x::xs' -> (insert x (ins_sort xs'))
```

Naturally, one can give a tail recursive of `ins_sort`

```
let ins_sort xs =
  let rec loop xs1 xs2 =
    match xs1 with
      [] -> xs2
    | x::xs1' -> loop xs1' (insert x xs2)
  in
    loop xs []
```

But the function `insert` can't be rewrite into a tail recursive definition. This is due to the fact that `insert` is commutative (`(insert x1 (insert x2 xs))` = `(insert x2 (insert x1 xs))`, but not the `::` constructor (in general `x1::x2::xs` ≠ `x2::x1::xs`).

Higher order functions The above sorting algorithm is a simple *iteration* of the inserting function over elements of the list to sort. This kind of iteration is generic over lists and can be recursively defined as a function of which one argument is the function to iterate

```
let rec it_list f a xs =
  match xs with
    [] -> a
  | x::xs -> (f x (it_list f a xs))
```

The first argument (`f`) is the function to iterate, the second argument (`a`) is the value for the base case and the third (`xs`) is the list over which iterate. The function `it_list` has type `('a -> 'b -> 'b) -> 'b -> 'a list -> 'b`. We have that `(it_list f a [x1; x2; ..;xn])` computes `(f x1 (f x2 .. (f xn a)..))`.

Using this iterator, we can defined `ins_sort` by

```
let ins_sort xs =
  it_list insert [] xs
```

The module `List` of the standard library provides some predefined iterators

```
val map : ('a -> 'b) -> 'a list -> 'b list

 List.map f [a1; ...; an] applies function f to a1, ..., an, and
 builds the list [f a1; ...; f an] with the results returned by f. Not
 tail-recursive.

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
 List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn.

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

 List.fold_right f [a1; ...; an] b is f a1 (f a2 (... (f an b)
 ...)). Not tail-recursive.

val for_all : ('a -> bool) -> 'a list -> bool

 List.for_all p [a1; ...; an] checks if all elements of the list satisfy
 the predicate p. That is, it returns (p a1) && (p a2) && ... && (p
 an).

val exists : ('a -> bool) -> 'a list -> bool

 List.exists p [a1; ...; an] checks if at least one element of the list
 satisfies the predicate p. That is, it returns (p a1) || (p a2) ||
 ... || (p an).
```

Our `it_list` is in fact `List.fold_right`.

With the `List.exists` iterator, it is possible to simply define the `mem` function

```
let mem x xs =
  let eq_x y =
    (y = x)
  in
    List.exists eq_x xs
```

As OCAML is a full functional language, there exist *functional expressions* which allow to write a function without to have to explicitely define it. For instance

```
let mem x xs =
  List.exists (fun y -> (y=x)) xs
```

where the expression `fun y -> (y=x)` is the *function* that checks if his argument `y` is equal to `x`.

Merge sort A less simple sorting algorithm is based on the principle of *"divide to conquer"*. It is the *mergesort*: to sort a list, split it in two parts, sort those two parts and *merge* the two sorted sublists.

To divide the list into two parts, we use the following

```
let divide n xs =
  let rec loop n xs xs1=
    match n, xs with
      0, _ -> (xs1, xs)
    | _, (x::xs') -> loop (n-1) xs' (x::xs1)
    | _ -> raise (Invalid_argument "divide")
  in
    loop n xs []
```

The `int` argument `n` gives the length of the first sublist. The order is not preserved in the first sublist but we don't care as it will be sorted latter.

Merging of two sorted lists will interleave the values of the two list such that the ordering is preserved. As it explores two lists, it is defined by *double induction*

```
let rec merge xs1 xs2 =
  match xs1, xs2 with
    [], _ -> xs2
  | _, [] -> xs1
  | (x1::xs1'), (x2::xs2') ->
     if x1 < x2 then
       x1::(merge xs1' xs2)
     else
       x2::(merge xs1 xs2')
```

The sorting function itself is

```
let merge_sort xs =
  let rec loop n xs =
    match xs with
      [] -> []
    | [x] -> xs
    | _ ->
      let n1 = n/2 in
      let n2 = n - n1 in
      let xs1, xs2 = divide n1 xs in
        merge (loop n1 xs1) (loop n2 xs2)
   in
     loop (List.length xs) xs
```

Note that we do so that the list is divided in two egal parts (more or less 1).

An alternative way of splitting lists is

```
let divide xs =
  let rec loop xs xs1 xs2 =
    match xs with
      [] -> (xs1, xs2)
    | [x] -> (x::xs1, xs2)
    | x1::x2::xs -> loop xs (x1::xs1) (x2::xs2)
  in
    loop xs [] []
```

with this, the sorting function becomes

```
let rec merge_sort xs =
  match xs with
    [] | [_] -> xs
  | _ ->
    let xs1, xs2 = divide xs in
      (merge (merge_sort xs1) (merge_sort xs2))
```

Remark how we have used a minimalist and compact pattern matching: we have join two cases with give a similar result; and we have avoid bindings by the use of the anonymous *universal pattern* (_).

The standard library of OCAML provides a tricky implementation of mergesort. It avoid to actually calculate the two parts of the divided lists. It is indeed desirable to avoid this because to get the first part of the list, as do our `divide` function, leads to build a *new copy* of this part. And this is repeated each time the dividing function is called. So, here it is how we can do better

We define a function that merely get the second part of the list

```
let rec tls n xs =
  match n, xs with
    0, _ -> []
  | n, x::xs -> tls (n-1) xs
  | _ -> raise (Invalid_argument "tls")
```

Remark that this function does not build any new structure.

We then use it this way in the sorting process

```
let rec merge_sort xs =
  let rec loop n xs =
    match n, xs with
      0, _ -> []
    | 1, x::_ -> [x]
    | _ ->
      let n1 = n/2 in
      let n2 = n - n1 in
        merge (loop n1 xs) (loop n2 (tls n2 xs))
  in
    loop (List.length xs) xs
```

The trick is that the first elements of the list are actually considered by the `loop` function itself while the last are given *via* the `tls` function.

Indeed, the implementation of the standard library optimizes things by considering, in the local recursive function, list of at least lenght 2, and moreover, it uses alternation of ordering to have a tail recursive merging process. But we will not consider it further.

1.2 User defined inductive types

ML languages are suitable for *symbolic computation* on *algebraic data types*. This is due to its ability of defining data structures by constructors (free symbols) and functions by pattern matching.

Binary trees For instance the set BT of binary trees (with labeled nodes) may be inductively defined by

- the *empty* tree belongs to BT;
- if x is a label and if b_1 and b_2 are two elements of BT then the tree whose root is labled x and the subtrees are b_1 and b_2 belongs to BT

Symbolicaly, let *Empty* be a constant and *Br* a ternary free symbol, we can set

- $Empty \in BT$
- if x is a label and $b_1, b_2 \in BT$ then $Br(x, b_1, b_2) \in BT$. We say that x is the *root label* and b_1, b_2 the *subtrees* (resp. *left* and *right*).

This definition can be rephrased in OCAML syntax

```
type 'a btree =
  Empty
| Node of ('a * 'a btree * 'a btree)
```

The key word `type` introduces the new data type definition. The name of the new datatype is `btree`. It is a generic data structure whose paramiter is `'a`. The two constructors cases are separeted by the vertical bar `|`. The two constructors name are `Empty` and `Node` and when it is a functional one, the type of its argument is given – *as a tuple*, if several values are needed – after the key word `of`. So defined data types are *sum types*. In the computer scientists litterature they are also called *variant type* as constructors may encapsulate values of any type.

The definition syntax of sum types intentionaly reminds the one of pattern matching because pattern matching is the intended method to define functions over values of sum types.

For instance, the membership function for trees can be defined as

```
let rec btree_mem v b =
  match b with
    Empty -> false
  | Node(x, b1, b2) -> (x = v) or (btree_mem v b1) or (btree_mem v b2)
```

Traversing binary trees *Traversing* a tree is a way to explore the values stored in it. There is three ways to do that we can illustrate as three ways to output the values of the labels

- *prefix*: output the root value, then traverse the left subtree, then traverse the right subtree;
- *postfix*: traverse the left subtree, then traverse the right subtree and, at the end, ouput the root value;
- *infix*: traverse the left subtree, then output the root value, then traverse the right subtree.

Those names come for the three ways to write propositional or arithmetics expressions.

Rather than ouputing values of labels, we propose to build the list of them according to the three traversing. Then will appear that the inductive structure of lists leads to more or less natural traversing functions.

So let's begin by *postfix* traversing. The most natural definition suggested by the above description is

```
let rec postfix1 b =
  match b with
      Empty -> []
    | Node(x, b1, b2) -> (postfix1 b1)@(postfix1 b2)@[x]
```

We don't like it because firstly it is not tail recursive and secondly the composition of lists appending leads to do several times the same work.

We can avoid to redo the same by patching our function this way

```
let postfix2 b =
  let rec loop b xs =
    match b with
        Empty -> xs
      | Node(x, b1, b2) -> loop b1 (loop b2 (x::xs))
  in
    loop b []
```

But this is still not tail recursive.

We can fix this by adding to the inner `loop` function an extra argument to push the right subtrees until the left one is consumed. We simulate so, but at lower cost, the pushing of the outer function call by the evaluation process

```
let postfix3 b =
  let rec loop b bs xs =
    match b, bs with
      Empty, [] -> xs
    | Empty, b::bs -> loop b bs xs
    | Node(x, b1, b2), bs -> loop b2 (b1::bs) (x::xs)
  in
    loop b [] []
```

Now we have no work redondancy and are tail recursive. We can improve a bit more our function by avoiding to push `Empty` subtrees. For this, we enforce the case analysis by deeper pattern matching

```
let postfix3' b =
  let rec loop b bs xs =
    match b, bs with
      Empty, [] -> xs
    | Empty, b::bs -> loop b bs xs
    | Node(x, Empty, Empty), [] -> (x::xs)
    | Node(x, Empty, Empty), b::bs -> loop b bs (x::xs)
    | Node(x, Empty, b2), bs -> loop b2 bs (x::xs)
    | Node(x, b1, Empty), bs -> loop b1 bs (x::xs)
    | Node(x, b1, b2), bs -> loop b2 (b1::bs) (x::xs)
  in
    loop b [] []
```

Let's now implement the *prefix* traversing. The natural recursive function is

```
let rec prefix1 b =
  match b with
      Empty -> []
    | Node(x, b1, b2) -> x::(prefix1 b1)@(prefix1 b2)
```

It has the two defaults of `postfix1`.

We can eliminate the work redoing in the same way, except that we put at the beginning what was put at the end

```
let rec prefix2 b =
  let rec loop b xs =
    match b with
        Empty -> xs
      | Node(x, b1, b2) -> x::(loop b1 (loop b2 xs))
  in
    loop b []
```

We could eliminate non tail recursivity as we did in *postfix3*

```
let prefix3_0 b =
  let rec loop b bs xs =
    match b, bs with
        Empty, [] -> xs
      | Empty, b::bs -> loop b bs xs
      | Node(x, b1, b2), bs -> loop b1 (b2::bs) (xs@[x])
  in
    loop b [] []
```

But doing so, we reintroduce work redondancy because of the list appending in `(xs@[x])`.

This can be improved by remarking that building a list by adding values at the end is equivalent to add all values at the head and then reverse the list. This give the definition

```
let prefix3 b =
  let rec loop b bs xs =
    match b, bs with
        Empty, [] -> List.rev xs
      | Empty, b::bs -> loop b bs xs
      | Node(x, b1, b2), bs -> loop b1 (b2::bs) (x::xs)
  in
    loop b [] []
```

It can also be improved by avoiding to push `Empty` trees (as an exercise).

We can't do better with the prefix traversing. This is due to the fact thaht in pur functional language, there is no way the have lists constructed by the end. Those structures are named *queues*. The inductive lists have naturaly the behaviour of *stacks*.

Let's now consider our third traversing: the *infix* one.

```
let rec infix1 b =
  match b with
      Empty -> []
    | Node(x, b1, b2) -> (infix1 b1)@[x]@(infix1 b2)
```

To step to a tail recursive version of this function, we push both the root label and the left subtree

```
let infix2 b =
  let rec loop b xbs xs =
    match b, xbs with
        Empty, [] -> xs
      | Empty, (x,b)::xbs -> loop b xbs (x::xs)
      | Node(x, b1, b2), xbs -> loop b2 ((x,b1)::xbs) xs
  in
    loop b [] []
```

The traversing implemented here is *rightmost* infix: the infix list for the right subtree is stored first in the accumulator `xs`, then is put the root label and then the list for the left subtree. We do this because of stack structures of lists.

For the infix traversing, we can also use a tricky way to "push" the pending computations by *rearranging* the tree structure. Indeed, a list `[x1; x2; ...; xk]` is almost a tree where lefts subtrees are always `Empty`: `Node(x1, Empty, Node(x2, Empty, ..., Node(xk, Empty, Empty))...)`. This gives, as a first attempt

```
let rec infix3 b =
  match b with
      Empty -> []
    | Node(x, Empty, b2) -> x::(infix3 b2)
    | Node(x1, Node(x2, b1, b2), b3) -> infix3 (Node(x2, b1, Node(x1, b2, b3)))
```

It is not tail recursive because of the second matching case.

Let's fix it. We cannot use an auxiliary list because of its stack structure which will reverse the order of elements. But we can simulate both the push of pending computations and values to be output by sliding to the left subtree the rearrangement

```
let infix3' b =
  let rec loop b xs =
    match b with
        Empty -> xs
      | Node(x, b1, Empty) -> loop b1 (x::xs)
      | Node(x1, b1, Node(x2, b2, b3)) -> loop (Node(x2, Node(x1, b1, b2), b3)) xs
  in
    loop b []
```

This operates the rightmost infix traversing.

Mutual recursion *General trees* are trees whose nodes may abitrarly branch: a node may have any number of subtrees. Such a data structure can be defined as the inductive sum type

```
type 'a gtree =
  GEmpty
| GNode of 'a * ('a gtree) list
```

We use the list structure (`('a gtree) list`) to have a variable number of subtrees attached to a node. Note that this entails that there is several values of type `'a gtree` which represent a tree with only one node: `GNode(x,[])`, `GNode(x,[GEmpty])`, `GNode(x,[GEmpty; GEmpty])`, etc.

A list of trees is called a `forest`. Functions over `gtree`'s often need to defined both a function for the tree and an other for the forest; and they often are *mutualy recursive*. For instance the prefix traversing of `gtree` is defined

```
let rec gprefix g =
  match g with
    GEmpty -> []
  | GNode(x, gs) -> x::(gprefixS gs)

and gprefixS gs =
  match gs with
    [] -> []
  | g::gs -> (gprefix g)@(gprefixS gs)
```

The key word `and` indicates the mutual dependence of the recursive definitions of the two functions. A mutual definition may involve an arbitrary number of functions.

Binary search trees As for the list case, the membership test can be improved, if the tree structure obey to an ordering. *Binary search trees* are values of `'a btree` which satisfy the following proprety:

- `Node(x, b1, b2)` is a binary search tree if `b1` and `b2` are binary search trees and the root label `x` is greater than the labels of `b1` and smaller than the labels of `b2`.
- `Empty` is a binary search tree.

The binary structure of the tree determines a partition of its labels edged by the root label. It then allows to proceed to *dichotomic* searching

```
let rec bstree_mem v b =
  match b with
    Empty -> false
  | Node(x, b1, b2) ->
     (v = x) or
     (if (v < x) then (bstree_mem v b1)
      else (bstree_mem v b2))
```

The ordered data structure of binary search trees can also be used as an intermediate *storage structure* for sorting lists. The sorting process is then divided into two steps

1. insert the values of the list to sort into a binary search tree.
2. traverse the tree structure in order to extract the ordered list.

To achieve the first step we define a function which inserts a new value, at its intended place, in a binary search tree

```
let rec ins_bstree v b =
  match b with
    Empty -> Node(v, Empty, Empty)
  | Node(x, b1, b2) ->
      if (v < x) then
        Node(x, (ins_bstree v b1), b2)
      else
        Node(x, b1, (ins_bstree v b2))
```

Note that the new value is always added as a new *leaf* of the tree.

The binary search tree structure is implemented by (a tail recursive) iteration of of insertion function

```
let bstree_of_list xs =
  let rec loop xs b =
    match xs with
      [] -> b
    | x::xs' -> loop xs' (ins_bstree x b)
  in
    loop xs Empty
```

The traversing which extracts the sorted list from a binary search tree is the infix one. Just call `infix` any function which implements it. The two steps of the sorting function is the *functional composition* of the two above functions.

```
let bstree_sort xs =
  infix (bstree_of_list xs)
```

Heapsort A *heap* is a *well balanced* binary tree such that the root label of any of its subtree is smaller[3] than all other labels of the subtree. A well balanced tree is one where each branch of the tree have the same length upto 1.

The *heapsort* proceeds also in two steps: build the heap structure then extract the sorted list. The key point of functional programs for heapsort is the function which adds a new element to a heap: it must preserve the "well balanced" property.

So, the key function is

```
let rec ins_heap x h =
  match h with
    Empty -> Node(x, Empty, Empty)
  | Node(x0, h1, h2) ->
      if (x < x0) then
        Node(x, h2, (ins_heap x0 h1))
      else
        Node(x0, h2, (ins_heap x h1))
```

Note that it is designed for minimal heaps: it keeps the smallest value as the root. The trick of this function is the rotation it operates with subtrees to preserve the "well balanced" property.

Storing values of the list to be sorted is done, as for the previous, by iterating the insertion function

```
let heap_of_list xs =
  let rec loop xs h =
    match xs with
      [] -> h
    | x::xs -> loop xs (ins_heap x h)
  in
    loop xs Empty
```

The extraction step traversing is a little less simple than the case of binary search trees. Indeed, the only think we know about heaps is that the root's value minimizes all the labels strored in the heap. So we will keep the root as first element of the extracted list. But we don't know any think about the relation between labels strored in the left subtree and the ones of the right subtree. So we need to *merge* the extracted sublist instead of simply appending them. This gives

```
let rec list_of_heap h =
  match h with
    Empty -> []
  | Node(x, h1, h2) -> x::(merge (list_of_heap h1) (list_of_heap h2))
```

Then the main function is

```
let heap_sort xs =
  list_of_heap (heap_of_list xs)
```

2 Proofs

Induction As the main paradigm of functional programs is recursion, a lot of proofs about functional programs require induction.

[3]Remplacing "smaller" by "greater" defines also a heap.

On the set of naturals $\mathbb{N}$, the induction principle can be given as

if $\Phi[0]$ and if $\forall y \in \mathbb{N}.\Phi[y] \Rightarrow \Phi[y+1]$ then $\forall x \in \mathbb{N}.\Phi[x]$

Considering +1 as the unary *successor* function, the set of naturals is recursively defined with the two *constructors* 0 and +1: $0 \in \mathbb{N}$ and if $x \in \mathbb{N}$ then $x+1 \in \mathbb{N}$. So, because it follow the constructor's structure of naturals, the above induction principle can be called *structural induction*. Extending this to inductive data types as lists or binary trees, we can set for them such structural induction principles. To adopt a "computer scientist" style, lets write `x` : `t` for `x` belongs to the set of values of type `t`. The structural induction on lists is

If Φ[`[]`] and if $\forall$`y` : `'a`.$\forall$`ys` : `'a list`.(Φ[`ys`] $\Rightarrow$ Φ[`y::ys`]) then $\forall$`xs` : `'a list`.Φ[`xs`]

And the structural induction on binary trees is

If Φ[`Empty`] and if $\forall$`x` : `'a`.$\forall$`b1` : `'a btree`.$\forall$`b2` : `'a btree`.(Φ[`b1`] $\wedge$ Φ[`b2`] $\Rightarrow$ Φ[`Node(x, b1, b2)`])
then $\forall$`b` : `'a btree`.Φ[`b`]

Note that the structural induction on lists follows the induction on naturals as the length of `[]` is 0 and the length of `x:xs` equals then length of `xs` plus 1. For the structural induction on binary trees we rather use a *generalized induction* principle using the ordering of naturals:

if $\forall y \in \mathbb{N}.\forall z \in \mathbb{N}.(z < y \ \wedge \ \Phi[z] \Rightarrow \Phi[y])$ then $\forall x \in \mathbb{N}.\Phi[x]$.

as the *size* – *ie* number of nodes – of `b1` and `b2` is strictly less than the size of `Node(x, b1, b2)`.

Types and values From the symbolic point of view, we can see the set of *values* as the set of ML expressions or *terms* build upon basic constants and constructors. So `true` belongs to the set of values of type `bool`, `123` belongs to the set of values of type `int` and `[1;2;3]` belongs to the set of values of type `int list`. This will be written `true` : `bool`, `123` : `int` and `[1;2;3]` : `int list`.

More generaly for the set of values of type `'a list`, what ever can be the type `'a` (and the set of its values), as we add with typing discipline, we have that if `x` : `'a` and if `xs` : `'a list` then `x::xs` : `'a list`. But recall that we give a stronger meaning to typing assignement. Namely here, it means that if `x` and `xs` are actually defined values then also is `x::xs`. This entails that the constructors of sum types are regarded as *totaly defined functions*. We will still use the typing like notation for this. For instance, we write `Node` : `('a * 'a btree * 'a btree) -> 'a btree`.

Equality statements When a function, say `f` : `t1 -> t2`, is applied to a value, say `v1` : `t1`, it is expected to result in a value `v2` : `t2`. We write this as an equational statement (`f v1`) = `v2`. For instance (`not true`) = `false`. Let `e1` and `e2` be two ML expressions where occur variables `x1 ...xk` with respective types `t1 ...tk`. The equality `e1 = e2` means that there exists a value `v` such that forall values `v1 ...vk` of respective types `t1 ...tk`, we have `e1`[`v1`/`x1` ... `vk`/`xk`] = `v` and `e2`[`v1`/`x1` ... `vk`/`xk`] = `v`. In other terms, the universal closure of `e1` = `e2` is valid. For instance (`false or x`) = `x`.

A point here is that the equality statement `e1` = `e2` can be true or false but can also be *neither true, neither false*: this is the case when `e1` or `e2` does not produce any value. Think, for instance to the incomplete definition of `fac` which may loop. The equality statement `e1` = `e2` has a truth value if and only if exists a type `t` such that `e1` : `t` and `e2` : `t`. So when this requirement is fullfilled, equality statements can be used as equations.

Symbolic evaluation The boolean disjunction can be defined using a `if-then-else` construction

```
let disj x1 x2 =
  if x1 then true else x2
```

By definition we have here that `(disj false x)` = `(if false then true else x)` which is given by substituing the actual arguments `false` and `x` to the formal parameters `x1` and `x2` in the body of the definition of `disj` (we call this *unfolding* the function's definition). We still want to have that `(disj false x)` = `x`. So we set the *semantics* of the `if-then-else` construction by putting the two equality statements:

- `(if true then e1 else e2)` = `e1`
- `(if false then e1 else e2)` = `e2`

One can also define the boolean disjunction using pattern matching

```
let disj x1 x2 =
  match x1 with
    true -> true
  | false -> x2
```

And we still want to have that `(disj false x)` = `x`. So we set, for the semantics of `match-with` construction applied to booleans

- `(match true with true -> e1 | false -> e2)` = `e1`
- `(match false with true -> e1 | false -> e2)` = `e2`

Those two semantics equations are valid, but they are far from a complete semantics of pattern matching, for instance, we also have

- `(match false with false -> e2 | true -> e1)` = `e2`
- `(match false with true -> e1 | _ -> e2)` = `e2`

We have remarked that pattern matching have also a binding effect. So, if the expression `e2` depends on a variable `x`, we have

- `(match false with true -> e1 | x -> e2)` = `e2[false/x]`

Moreover, the `match-with` construction may adress any *value*, in the sens given at the begining of this paragraph. For instance, basic pattern matching on lists is

- `match [] with [] -> e1 | x::xs -> e2` = `e1`
- `match e::es with [] -> e1 | x::xs -> e2` = `e2[e/x, es/xs]`

We will not go further with a formal definition of the semantics of pattern matching. Just keep in mind that it mimics the evaluation process of `match-with` construction.

We call *symbolic evaluation* the application of user defined functions unfolding and semantics equations of language constructs.

Here is for instance how the concatenation of lists is defined and and its application evaluated

```
let rec (@) xs1 xs2 =
  match xs1 with
    [] -> xs2
  | x::xs1' -> x::(xs1'@xs2)
```

```
[1]@[3;4]  =  (1::[])@[3;4]
           =  match (1::[]) with [] -> [3;4] | x::xs1' -> x::(xs1'@[3;4])
           =  1::([]@[3;4])
           =  1::(match [] with [] -> [3;4] | x::xs1' -> x::(xs1'@[3;4]))
           =  1::[3;4]
           =  [1;3;4]
```

More generaly we can say that, using symbolic evaluation, we get the two equalities:

- `([]@xs2)` = `xs2`
- `((x1::xs1)@xs2)` = `x1::(xs1@xs2)`

which is closed to the equaltionnal way to define the concatenation of lists.

As a first step in proving programs, here are two little properties on the append function `@`. We will prove them using symbolic evaluation and structural induction.

LEMMA(append1):

$$\forall \texttt{x} : \texttt{'a}.\forall \texttt{xs} : \texttt{'a list}.(\texttt{([x] @ xs)} = \texttt{(x::xs)})$$

Recall that `[x]` is a short cut for `x::[]`.

PROOF: by symbolic evaluation.

```
((x::[]) @ xs)  =  match x::[] with [] -> xs | x'::xs' -> x'::(xs'@xs)
                =  x::([]@xs)
                =  x::(match [] with [] -> xs | x'::xs' -> x'::(xs'@xs))
                =  x::xs
```

LEMMA(appendNil):

$$\forall \texttt{xs} : \texttt{'a list}.(\texttt{(xs @ [])} = \texttt{xs})$$

The concatenation is recursively defined by case on its first argument, so

PROOF: by structural induction on `xs`.

- If `xs` equals `[]`, by symbolic evaluation, we have `([]@[])` = `[]`.

- If `xs` equals `x::xs'`, assuming the induction hypothesis `(xs' @ [])` = `xs'`, we have to prove that `(x::xs') @ []` = `(x::xs')`. Actually

```
(x::xs') @ []  =  x::(xs' @ [])   by symbolic evaluation
               =  (x::xs')        by induction hypothesis
```

We left as an exercise to the reader to prove that the concatenation of lists is associative:

LEMMA(assoc. append):

$$\forall \texttt{xs1, xs2, xs3} : \texttt{'a list}.(\texttt{xs1@(xs2@xs3)} = \texttt{(xs1@xs2)@xs3})$$

Equality of programs In functional programming, equality of programs is equality of functions. Two functions are equal when they give the same values for the same arguments.

We can so prove that our two programs `rev1` and `rev2` both compute the same function (in the mathematical sens). This is usefull at the practical level to ensure that the improved tail recursive implementation of reversing a list actually correspond to a less performent but more natural implementation. Lot of bugs are due to such attempt to improve the behaviour of a program which breaks the safety of a former version of a program.

So, let's recall the definitions

```
let rec rev1 xs =
  match xs with
    [] -> []
  | x::xs -> (rev1 xs)@[x]

let rec rev2 xs =
  let rec loop xs1 xs2 =
    match xs1 with
      [] -> xs2
    | x::xs1' -> loop xs1' (x::xs2)
  in
    loop xs []
```

We want to establish the following

THEOREM:

$$\forall \texttt{xs:'a list}.((\texttt{rev1 xs}) = (\texttt{rev2 xs}))$$

We assume that `rev1` : `'a list -> 'a list` and `rev2` : `'a list -> 'a list` have been established.

The inner `loop` of `rev2` uses an accumulator (its second argument). This feature gives to the local `loop` function a properties that is more *general*, regarding its relationship with `rev1` than the one set by our theorem. We call this property the *invariant* of the local function `loop` from which we will deduce our theorem. So, let's prove first that

LEMMA(key lemma):

$$\forall \texttt{xs1, xs2:'a list}.(\texttt{rev2.loop xs1 xs2}) = (\texttt{rev1 xs})\ \texttt{@ xs2}$$

(Note: we use the doted notation `rev2.loop` to designate the function `loop` local to the definition of `rev2`)

PROOF: we prove by structural induction on `xs1:'a list` that

$$\forall \texttt{xs2:'a list}.(\texttt{rev2.loop xs1 xs2}) = (\texttt{rev1 xs})\ \texttt{@ xs2}$$

Note: the universal quatification on `xs2` is required to have a suitable induction hypothesis.

- If `xs1` equals `[]`, `(rev2.loop [] xs2)` = `xs2` = `[] @ xs2` = `(rev1 []) @ xs2`.

- If `xs1` equals `x::xs1'`, our induction hypothesis is

$$\forall \texttt{xs2:'a list}.(\texttt{rev2.loop xs1' xs2}) = (\texttt{rev1 xs'})\ \texttt{@ xs2}$$

Then we have

`(rev2.loop (x::xs1') xs2)`	=	`(rev2.loop xs1' (x::xs2))`	by sym. eval.
	=	`(rev1 xs1') @ (x::xs2)`	by induction hypothesis[4]
	=	`(rev1 xs1') @ ([x] @ xs2)`	by lemma (append1)
	=	`((rev1 xs1') @ [x]) @ xs2`	by assoc. append
	=	`(rev1 (x::xs1')) @ xs2`	by sym. eval.

PROOF(of the theorem):

`(rev2 xs)`	=	`(rev2.loop xs [])`	by def.
	=	`(rev1 xs) @ []`	by (key lemma)
	=	`(rev1 xs)`	by lemma (appendNil)

Double induction The `merge` function which explore in parallel two lists is defined with a double recursion. Proofs about it may involve a double induction. To simplify, here is the proof that `merge` is totally defined.

THEOREM:

$$\forall \texttt{xs1, xs2} : \texttt{'a list}.(\texttt{(merge xs1 xs2)} : \texttt{'a list})$$

Note: recall that we use `e` : `t` *to mean that the value of* `e` *belongs to the set of values of the type* `t`*. So, the atomic formula* (`(merge xs1 xs2)` : `'a list`) *indeed means that* `merge` *is a well defined list value for all list values* `xs1` *and* `xs2`*.*

PROOF: by induction on `xs1` we prove that

$$\forall \texttt{xs2} : \texttt{'a list}.(\texttt{(merge xs1 xs2)} : \texttt{'a list})$$

- If `xs1` equals `[]`, we assume that `xs2` : `'a list`. By symbolic evaluation, we have that `(merge [] xs2)` = `xs2`. Then, by hypothesis, we have that (`(merge [] xs2)` : `'a list`).

- If `xs1` equals `x1::xs1'`. Under the following hypothesis (`x1 : 'a`), (`xs1' : 'a list`) and (induction hypothesis) ∀`xs2` : `'a list`.(`(merge xs1' xs2)` : `'a list`) we prove (`(merge x1::xs1' xs2)` : `'a list`) by induction on `xs2`

– If `xs2` equals `[]`, as `(merge x::xs1' [])` = `x::xs1'` we have by hypothesis and the cons (`::`) typing property that (`(merge x::xs1' [])` : `'a list`).

– If `xs2` equals `x2::xs2'`, we have a second induction hypothesis: (`(merge x1::xs1' xs2')` : `'a list`). By symbolic evaluation, we have that

```
(merge x1::xs1' x2::xs2') =
  if (x1 < x2) then x1::(merge xs1' x2::xs2') else x2::(x1::xs1' xs2')
```

We reason by case on the boolean value of `(x1 < x2)`:

— If `(x1 < x2)` equals `true` we have that (`x1::(merge xs1' x2::xs2')` : `'a list`) by our first induction hypothesis.

— If `(x1 < x2)` equals `false` we have that (`x2::(x1::xs1' xs2')` : `'a list`) by our second induction hypothesis.

In fact, the double induction here and the use of the two induction hypothesis is the unfolding of lexicographic ordering on the pair of arguments.

Nested double induction The infix traversing implemented by rearrangement of the tree in the function `infix3'` is defined by nested recursion: one first on the tree and, in the `Node` case, a second on one of its subtree. This computational trick has a counter part when reasonning by induction on such programs: the formula for the second induction embed the first induction hypothesis.

Let's see this in the proof of the equality between the naive infix traversing implentation (function `infix1`) and the more structural `infix3'`.

THEOREM:

$$\forall \texttt{b} : \texttt{'a btree}.\texttt{(infix3' b)} = \texttt{(infix1 b)}$$

Here again, the theorem will be a consequence of an invariant property of the local `loop` in the `infix3'` definition.

LEMMA:

$$\forall \texttt{b} : \texttt{'a btree}.\forall \texttt{xs} : \texttt{'a list}.\texttt{(infix3'.loop b xs)} = \texttt{(infix1 b)@xs}$$

PROOF: by structural induction on `b`.

- If `b` equals `Empty`, immediate by sym. eval.

- If `b` equals `Node(x1, b1, b2)`, we will prove that

$$\forall \texttt{x:'a}.\forall \texttt{b1 : 'a btree}.$$
$$(\forall \texttt{xs : 'a list}.((\texttt{infix3'.loop b1 xs}) = (\texttt{infix1 b1})\texttt{@xs})$$
$$\Rightarrow (\forall \texttt{xs : 'a list}.(\texttt{infix3'.loop (Node(x, b1, b2)) xs}) = (\texttt{infix (Node(x1, b1, b2))})\texttt{@xs}))$$

by induction on `b2`

– If `b2` equals `Empty`. Assuming $\forall$`xs : 'a list.(infix3'.loop b1 xs) = (infix1 b1)@xs` we prove

$$(\texttt{infix3'.loop (Node(x1, b1, Empty)) xs}) = (\texttt{infix1 (Node(x1, b1, Empty))})\texttt{@xs}$$

That is to say `(infix3'.loop b1 (x1::xs)) = (infix1 b1)@[x1]@(infix1 Empty)@xs` Which is true by hypothesis and sym. eval.

– If `b2` equals `Node(x2, b2, b3)`. Let's assume the two induction hypothesis

$HR1$:
$\forall$`x:'a.`$\forall$`b1 : 'a btree.`
$((\forall$`xs : 'a list.(infix3'.loop b1 xs) = (infix1 b1)@xs`$)$
$\Rightarrow (\forall$`xs : 'a list.(infix3'.loop (Node(x, b1, b2)) xs) = (infix1 (Node(x, b1, b2))@xs`$)))$
$HR2$:
$\forall$`x:'a.`$\forall$`b1 : 'a btree.`
$((\forall$`xs : 'a list.(infix3'.loop b1 xs) = (infix1 b1)@xs`$)$
$\Rightarrow (\forall$`xs : 'a list.(infix3'.loop (Node(x, b1, b3)) xs) = (infix1 (Node(x, b1, b3))@xs`$)))$
and also
H : `(infix3'.loop b1 xs) = (infix1 b1)@xs)`

We have then to prove

```
(infix3'.loop (Node(x1, b1, Node(x2, b2, b3)) xs)
  = (infix1 (Node(x1, b1, Node(x2, b2, b3))))@xs
```

That is to say, by definition of `infix3'`

```
(infix3'.loop (Node(x2, Node(x1, b1, b2), b3)) xs)
  = (infix1 (Node(x1, b1, Node(x2, b2, b3))))@xs
```

By definition of `infix1` and associativity of `@` we have that

```
(infix1 (Node(x1, b1, Node(x2, b2, b3))))@xs
  = (infix1 (Node(x2, Node(x1, b1, b2 ), b3)))@xs
```

So, we need to prove that

```
(infix3'.loop (Node(x2, Node(x1, b1, b2), b3)) xs)
  = (infix1 (Node(x2, Node(x1, b1, b2 ), b3)))@xs
```

From $HR1$ and H we get

$F1$: `(infix3'.loop (Node(x1, b1, b2)) xs) = (infix1 (Node(x1, b1, b2))@xs)`

Then, from $HR2$ and $F1$ we get the expected

```
(infix3'.loop (Node(x2, Node(x1, b1, b2), b3)) xs)
  = (infix1 (Node(x2, Node(x1, b1, b2 ), b3)))@xs
```

Specification We want now to scale to the proof that some program meets some intended specification. For instance, that a program computes a *sorted permutation* of its input. For this we first need to set the *specification* of the intended behaviour of the programm and secondly prove the *correctness* of the functions involved in the program.

Intuitively, a sorted list is such that for any values `x1`, `x2` in the list, if `x1` is *before* `x2` then $\texttt{x1} \leq \texttt{x2}$. Because of transitivity of the ordering $\leq$, we only need to consider elements `x1` and `x2` such that `x1` is *just before* `x2`. This is easy to set if `x1` and `x2` are the two first elements of the list. In this case, by construction, we know that the list has the form `x1::x2::xs` for some `xs`. We can then define a *recursive function* which checks if a list is sorted or not

```
let rec sorted xs =
  match xs with
    [] -> true
  | [x] -> true
  | x1::x2::xs -> (x1 <= x2) && (sorted (x2::xs))
```

Alternatively, we can define the property to be sorted as the *inductive predicate Sorted* satisfying the three axioms

DEFINITION(inductive predicate *Sorted*):

1. $\mathit{Sorted}(\texttt{[]})$
2. $\forall \texttt{x} : \texttt{'a}.\mathit{Sorted}(\texttt{[x]})$
3. $\forall \texttt{x1}, \texttt{x2}; \texttt{'a}.\forall \texttt{xs} : \texttt{'a}.(\texttt{x1} \leq \texttt{x2} \ \wedge \ \mathit{Sorted}(\texttt{x2::xs}) \equiv \mathit{Sorted}(\texttt{x1::x2::xs}))$

where $\texttt{x1} \leq \texttt{x2}$ is a shortcut for $\texttt{(x1 <= x2)} = \texttt{true}$.

With such a structural definition of the intended proprety, it is easy to prove the correctness of the sorting function `ins_sort`.

THEOREM:

$$\forall \texttt{xs} : \texttt{'a list}.\mathit{Sorted}(\texttt{ins\_sort xs})$$

This theorem is a mere consequence of the fact that the inserting function preserves the ordering.

LEMMA:

$$\forall \texttt{xs} : \texttt{'a list}.\forall \texttt{x} : \texttt{'a}.(\mathit{Sorted}(\texttt{xs}) \Rightarrow \mathit{Sorted}(\texttt{(ins x xs)}))$$

PROOF: by structural case on `xs`.
(*Note: "structural case" is a degenerated structural induction where induction hypothesis are ignored.*)

- If `xs` equals `[]`, the result is immediate.

- If `xs` equals `x1::xs1`, we have to prove that for some $\texttt{x0} : \texttt{'a}$, we have

$$\mathit{Sorted}(\texttt{x1::xs1}) \Rightarrow \mathit{Sorted}(\texttt{(ins x0 x1::xs1)})$$

By definition of `ins` we have that

$$\texttt{(ins x0 x1::xs1)} = \texttt{if (x0 <= x1) then x0::x1::xs1 else x1::(ins x0 xs1)}$$

So, let's consider the two cases:

– If $\texttt{x0} \leq \texttt{x1}$, assuming $\mathit{Sorted}(\texttt{x1::xs1})$, we need $\mathit{Sorted}(\texttt{x0::x1::xs1})$. What we have by *Sorted*.3.

– If $\texttt{x1} \leq \texttt{x0}$, $\texttt{(ins x0 x1::xs1)} = \texttt{x1::(ins x0)}$. We deduce $\mathit{Sorted}(\texttt{x1::(ins x0)})$ from the most general

$$\forall \texttt{x} : \texttt{'a}.(\texttt{x} \leq \texttt{x0}, \mathit{Sorted}(\texttt{x::xs1}) \Rightarrow \mathit{Sorted}(\texttt{x::(ins x0 xs1)}))$$

that we prove by induction on `xs1`.

— If `xs1` equals `[]`, we have `x::(ins x0 [])` = `x::x0::[]` and, as `x` ≤ `x0` by hypothesis, it is sorted.

— If `xs1` equals `x2::xs1`. Assume, for some `x1`, that `x1` ≤ `x0`, *Sorted*(`x1::x2::xs1`). This gives, by *Sorted*.3, `x1` ≤ `x2` and *Sorted*(`x2::xs1`). Remains to prove *Sorted*(`x1::(ins x0 x2::xs1)`).

According to the definition of `ins`, we consider for this the two cases:

– – If `x0` ≤ `x2`, we have to prove *Sorted*(`x1::x0::x2:xs1`), which we have as `x1` ≤ `x0` ≤ `x2` and *Sorted*(`x2::xs1`).

– – If `x2` ≤ `x0`, we have to prove *Sorted*(`x1::x2::(ins x0 xs1)`). As we have `x1` ≤ `x2`, we only need *Sorted*(`x2::(ins x0 xs1)`) which we have by induction hypothesis (here is why we needed to generalize `x` before doing induction on `xs1`).

To acheive the correctness of `ins_sort` we have to establish that it computes a permutation of its argument. We consider that a list is a permutation of an other if and only if the number of occurences of any value `x` in the first list is equal to its number of occurences in the second. So, we define the (ML) function which counts the number of occurences of an `x` in a list:

```
let rec nb x xs =
  match xs with
    [] -> 0
  | x'::xs' ->
      if (x = x') then 1 + (nb x xs')
      else (nb x xs')
```

Note: we don't want to have an optimized or a tail recursive definition of this function as we don't want to use it for computation but rather as a specification. And for this purpose a direct and simple expression is better.

We will need the following trivial fact

LEMMA(nb-cons):

$$\forall \texttt{x1, x2} : \texttt{'a}.\forall \texttt{xs1, xs2} : \texttt{'a list}.$$
$$((\texttt{nb x1 xs1}) = (\texttt{nb x1 xs2}) \Rightarrow (\texttt{nb x1 x2::xs1}) = (\texttt{nb x1 x2::xs2}))$$

PROOF: trivial by symbolic evaluation and case analysis of `x1` = `x2`.

The correctness property we want is

THEOREM:

$$\forall \texttt{xs} : \texttt{'a list}.\forall \texttt{x} : \texttt{'a}.(\texttt{nb x xs}) = (\texttt{nb x (ins\_sort xs)})$$

It is a consequence of

LEMMA:

$$\forall \texttt{xs} : \texttt{'a list}.\forall \texttt{x1, x2} : \texttt{'a}.(\texttt{nb x1 x2::xs}) = (\texttt{nb x1 (ins x2 xs)})$$

PROOF: by structural induction on `xs`.

- If `xs` equals `[]`, it is immediate as `(ins x2 [])` = `x2::[]`.

- If `xs` equals `x3::xs`. By definition of `ins` we have two cases:

– If `(ins x2 x3::xs)` = `x2::x3::xs` then the result is immediate.

– If `(ins x2 x3::xs)` = `x3::(ins x2 xs)`, by induction hypothesis, `(nb x1 x2::xs)` = `(nb x1 (ins x2 xs))` and it is clear that in this case `(nb x1 x2::x3::xs)` = `(nb x1 x3::(ins x2 xs))`.

Tuning specification The proof that insertion sort computes a sorted list comes naturaly by structural induction because both the program and the definition of *Sorted* use simple structural tools. This is not as simple with mergesort, because of the merging function.

Indeed, a proof that `merge_sort` is correct requires to prove that the merge function preserves the ordering

LEMMA(Sorted-merge):

$$\forall \texttt{xs1, xs2} : \texttt{'a list}.(\mathit{Sorted}(\texttt{xs1}) \;\wedge\; \mathit{Sorted}(\texttt{xs2}) \Rightarrow \mathit{Sorted}(\texttt{merge xs1 xs2}))$$

Trying to attack directly the proof of this lemma by double structural induction will lead to multiply case analysis. Let's outline why: after double induction, there is two cases of `merge` definition `x1::(merge xs1 x::xs2)` and `x2::(merge x1::xs1 xs2)`; but this is not enough to use the $S3$ clause of *Sorted*'s definition. So in each case for `merge` we have to apply an other case analysis (resp. on `xs1` and `xs2`). This gives the total of 10 cases splitting. So we are tempted to reduce this by setting an alternative characterisation of being sorted as *"adding a smaller element at the head of a sorted list gives a sorted list"*.

So, we need first to extend the ordering to a relation between a value `x` and a list `xs`. We write $\texttt{xs} \gg \texttt{x}$ to mean that all values of `xs` are greater than `x`, which reads also `x` is smaller than all values in `xs`[5]. To define this relation, think about the recursive boolean function you would write to check that a value `x` is smaller than all value in `xs` and turn it to the inductive definition of a predicate. You have graet chances to choose the following

DEFINITION($\texttt{xs} \gg \texttt{x}$): where `xs` : `'a list` and `x` : `'a`

1. $\forall \texttt{x} : \texttt{'a}.\texttt{[]} \gg \texttt{x}$
2. $\forall \texttt{x, x0} : \texttt{'a}.\forall \texttt{xs} : \texttt{'a list}.((\texttt{x0} \geq \texttt{x}) \;\wedge\; (\texttt{xs} \gg \texttt{x}) \equiv (\texttt{x0::xs} \gg \texttt{x}))$

We leave to the reader the definition of the dual $\texttt{xs} \ll \texttt{x}$.

Using this we can now give a new way to get recursively non empty sorted lists (the empty case does not need any improvement). We set it as the following theorem to ensure that our original view is preserved

THEOREM(Sorted-cons):

$$\forall \texttt{x} : \texttt{'a}.\forall \texttt{xs} : \texttt{'a list}.(\mathit{Sorted}(\texttt{xs}) \;\wedge\; \texttt{xs} \gg \texttt{x} \Leftrightarrow \mathit{Sorted}(\texttt{x::xs}))$$

PROOF: immediate by case on `xs`.

We must prove that `merge` does not introduce noise and preserves the $\gg$ relation.

LEMMA(merge-$\gg$):

$$\forall \texttt{x0} : \texttt{'a}.\forall \texttt{xs1, xs2} : \texttt{'a list}.(\texttt{xs1} \gg \texttt{x0} \;\wedge\; \texttt{xs2} \gg \texttt{x0} \Rightarrow (\texttt{merge xs1 xs2}) \gg \texttt{x0})$$

PROOF: by double induction on `xs1` and `xs2`

- If `xs1` or `xs2` equal `[]`, trivial.

- If `xs1` equals `x1::xs1` and `xs2` equals `x2::xs2`, according to cases of $\texttt{x1} \leq \texttt{x2}$

– If $\texttt{x1} \leq \texttt{x2}$, we have $\texttt{x1} \geq \texttt{x0}$ by the hypothesis $\texttt{x1::xs1} \gg \texttt{x0}$ and we have $(\texttt{merge xs1 x2::xs2}) \gg \texttt{x0}$ from the first induction hypothesis, then $\texttt{x1::(merge xs1 x2::xs2)} \gg \texttt{x0}$.

– If not, we have $\texttt{x2} \geq \texttt{x0}$ by the hypothesis $\texttt{x2::xs2} \gg \texttt{x0}$ and we have $(\texttt{merge x1::xs1 xs2}) \gg \texttt{x0}$ by the second induction hypothesis, then $\texttt{x2::(merge x1::xs1 xs2)} \gg \texttt{x0}$.

PROOF(of lemma (Sorted-merge)): by double induction on `xs1` and `xs2`

[5] We should have written as well $\texttt{x} \ll \texttt{xs}$. The present choice is guided by what follows about binary search trees.

- The case where `xs1` or `xs2` equal `[]` is still trivial.

- So let `xs1` equals `x1::xs1` and `xs2` equals `x2::xs2`. According to the definiton of merge, there is two cases.

– If $\texttt{x1} \leq \texttt{x2}$, we have two prove $Sorted$(`x1::(merge xs1 x2::xs2)`). We get $Sorted$(`merge xs1 x2::xs2`) by induction hypothesis. So, by (Sorted-cons), we need to we prove that `(merge xs1 x2::xs2)` $\gg$ `x1`.

We have by hypothesis that $Sorted$(`x1::xs1`), so `xs1` $\gg$ `x1`; we are in the case where $\texttt{x1} \leq \texttt{x2}$ (*ie* $\texttt{x2} \geq \texttt{x1}$) and we have `xs2` $\gg$ `x2` from the hypothesis that $Sorted$(`x2::xs2`), so `x2::xs2` $\gg$ `x1`; then, by lemma (merge-$\gg$) we have `(merge xs1 x2::xs2)` $\gg$ `x1`.

– The proof that $Sorted$(`x2::(merge x1::xs1 xs2)`) follows a similar reasonning line.

Traversing structure The sorting algorithm using intermediate binary search tree structure relies on two steps

1. the spliting of the value to sort into two globaly ordered parts.
2. the view of sorted lists as the concatenation of already sorted lists.

The spliting correspond in fact to the definition of binary search trees: what is on the left is less than the root label; what is on the right is more. All we need can be defined on the inductive structure of `'a btree`'s.

Definitions of *"to be less than"* and *"to be more than"* are

DEFINITION($\texttt{b} \ll \texttt{x}$): where `b` : `'a btree` and `x` : `'a`

- $\texttt{Empty} \ll \texttt{x}$
- $\forall \texttt{x0} : \texttt{'a}.\forall \texttt{b1}, \texttt{b2} : \texttt{'a btree}.(\texttt{b1} \ll \texttt{x} \wedge \texttt{x0} \leq \texttt{x} \wedge \texttt{b2} \ll \texttt{x} \equiv \texttt{Node(x0, b1, b2)} \ll \texttt{x})$

DEFINITION($\texttt{b} \gg \texttt{x}$): where `b` : `'a btree` and `x` : `'a`

- $\texttt{Empty} \gg \texttt{x}$
- $\forall \texttt{x0} : \texttt{'a}.\forall \texttt{b1}, \texttt{b2} : \texttt{'a btree}.(\texttt{b1} \gg \texttt{x} \wedge \texttt{x0} \geq \texttt{x} \wedge \texttt{b2} \gg \texttt{x} \equiv \texttt{Node(x0, b1, b2)} \gg \texttt{x})$

Note: we overload *the symbols $\ll$ and $\gg$ that we used for lists. But as all our variables will be typed, this should not cause confusion for the reader and emphasis the relationship of the minimizing/maximizing relations.*

A the definition of *"being a binary search tree"* is

DEFINITION(Bst(`b`)): where `b` : `'a btree`

1. $Bst(\texttt{Empty})$
2. $\forall \texttt{x0} : \texttt{'a}.\forall \texttt{b1}, \texttt{b2} : \texttt{'a btree}.(Bst(\texttt{b1}) \wedge \texttt{b1} \ll \texttt{x0} \wedge Bst(\texttt{b2}) \wedge \texttt{b2} \gg \texttt{x0} \equiv Bst(\texttt{Node(x0, b1, b2)}))$

To ensure the correctness of the first step of binary search tree sorting, we must prove that the insertion function `ins_bstree` preserve the Bst property

LEMMA(Bst-`ins_bstree`):

$$\forall \texttt{b} : \texttt{'a btree}.\forall \texttt{x} : \texttt{'a}.(Bst(\texttt{b}) \Rightarrow Bst(\texttt{ins\_bstree x b}))$$

Structural induction on b will give by induction hypothesis that inserting in left or right subtrees produce binary search trees. But it will not ensure that the $\ll$ and $\gg$ relations are satisfied. So it must be proved that insertion of smaller or greater values does not break the $\ll$ or $\gg$ relation.

LEMMA:(ins_bstree-$\ll$):

$$\forall \texttt{x1, x2} : \texttt{'a}.\forall \texttt{b} : \texttt{'a btree}.(\texttt{b} \ll \texttt{x2} \ \wedge \ \texttt{x1} \leq \texttt{x2} \Rightarrow (\texttt{ins\_bstree x1 b}) \ll \texttt{x2})$$

PROOF: immediate by induction on b

LEMMA:(ins_bstree-$\gg$):

$$\forall \texttt{x1, x2} : \texttt{'a}.\forall \texttt{b} : \texttt{'a btree}.(\texttt{b} \gg \texttt{x2} \ \wedge \ \texttt{x1} > \texttt{x2} \Rightarrow (\texttt{ins\_bstree x1 b}) \gg \texttt{x2})$$

PROOF: immediate by induction on b

PROOF(Bst-ins_bstree): by induction on b with the help of lemmas (ins_bstree-$\ll$) and (ins_bstree-$\gg$) for the inductive case.

Now comes the second step of the algorithm: the extraction of the sorted list from the tree structure. The proof that the infix traversing of a binary search tree gives a sorted lists relies on two propreties:

- the fact that the relations $\ll$ and $\gg$ are carried from trees to lists
- a new view of sorted lists as the concatenation of sorted lists

Those two properties need the generalisation of $\ll$ and $\gg$ relation to list concatenation.

LEMMA(append-$\ll$):

$$\forall \texttt{xs1}, \texttt{xs2} : \texttt{'a list}.\forall \texttt{x} : \texttt{'a}.(\texttt{xs1} \ll \texttt{x} \ \wedge \ \texttt{xs2} \ll \texttt{x} \Rightarrow \texttt{xs1@xs2} \ll \texttt{x})$$

PROOF: immediate by induction on xs1

LEMMA(append-$\gg$):

$$\forall \texttt{xs1}, \texttt{xs2} : \texttt{'a list}.\forall \texttt{x} : \texttt{'a}.(\texttt{xs1} \gg \texttt{x} \ \wedge \ \texttt{xs2} \gg \texttt{x} \Rightarrow \texttt{xs1@xs2} \gg \texttt{x})$$

PROOF: immediate by induction on xs1

The first property which carry $\ll$ and $\gg$ from trees to lists, is expressed by the two lemmas

LEMMA(btree-list-$\ll$):

$$\forall \texttt{b} : \texttt{'a btree}.\forall \texttt{x} : \texttt{'a}(\texttt{b} \ll \texttt{x} \Rightarrow (\texttt{infix b}) \ll \texttt{x})$$

PROOF: by structural induction on b with the help of ((append-$\ll$).

LEMMA(btree-list-$\gg$):

$$\forall \texttt{b} : \texttt{'a btree}.\forall \texttt{x} : \texttt{'a}(\texttt{b} \gg \texttt{x} \Rightarrow (\texttt{infix b}) \gg \texttt{x})$$

PROOF: by structural induction on b with the help of ((append-$\gg$).

The second property, which see sorted as the concatenation of separated sorted lists is given by

LEMMA(Sorted-append)

$$\forall \texttt{xs1}, \texttt{xs2} : \texttt{'a list}.\forall \texttt{x} : \texttt{'a}.(Sorted(\texttt{xs1}) \ \wedge \ \texttt{xs1} \ll \texttt{x} \ \wedge \ Sorted(\texttt{xs2}) \ \wedge \ \texttt{xs2} \gg \texttt{x} \Rightarrow Sorted(\texttt{xs1@[x]@xs2}))$$

*Note: the expression **xs1@[x]@xs2** correspond to one particular definition of the infix traversing (namely, our function **infix1**). It does not mean that we are forced to use this function to define binary search tree* *sorting. We can use any other implementation of infix traversing provided that we have proved the equality between our function and **infix1** which is used here as the specification of infix traversing.*

PROOF: by induction on `xs1`

– If `xs1` equals `[]`, as *Sorted*(`xs2`) and `xs2` $\gg$ `x` by hypothesis, we have *Sorted*(`[x]@xs2`) by (Sorted-cons).

– If `xs1` equals `x1::xs1`, from the induction hypothesis we deduce *Sorted*(`xs1@[x]@xs2`). We will have *Sorted*(`x1::xs1@[x]@xs2`) if we prove that `xs1@[x]@xs2` $\gg$ `x1`.
We have `xs1` $\gg$ `x1` from the hypothsesis *Sorted*(`x1::xs1`); we have `x1` $\leq$ `x` (*ie* `x` $\geq$ `x1`) from the hypothesis `x1::xs1` $\ll$ `x`; and then we have `xs2` $\gg$ `x1` from the hypothesis `xs2` $\gg$ `x` and the previous `x` $\geq$ `x1`. What is implicit in this is that the separating `x` gives that values in `xs1` are all smaller than the one in `xs2`.