INF231: Functional Algorithmic and Programming

Lecture 5: Lists

Academic Year 2020 - 2021

About Lists

Some motivation

So far data (handled by functions) are simple: values of some (complex) type
$\hookrightarrow$ how to manipulate an arbitrary number of values (of a given type)?

List are useful in modelling

Example (What can be modelled using lists)

- students of a class
- grades of a student
- the hand in a card-game

Lists have a special status in computer science:

- often used (useful in modelling)
- easy to manipulate (simple basis operations + library of complex operations)

Lists are first-class citizens in OCaml (contrarily to C).

Defining lists

What is a list?

- a finite series of values of the same type
- arbitrary length
- the order between its elements matters

Definition (Inductive ("recursive") definition of lists)

Given a set E, the set of lists over E is the largest set s.t.:

1. it contains a basis element: nil
2. given a list l and $e \in E$, $cons(e, l)$ is a list over E

Type `List` is a recursive union type:

1. A symbolic constant representing the empty list: Nil
2. A constructor, to "append an element to an existing list": Cons

Remark It differs from enumerated, product, and union types □

Syntax

Given some *existing* type `t`:

```
type list_of_t = Nil | Cons of t * list_of_t
```

The list where elements are `v1`, `v2`, ..., `vn` (in this order) is noted:

```
Cons (v1, Cons (v2, ...,Cons (vn, Nil) ...))
```

More generally, elements of a list can be arbitrary expressions:

```
Cons (expr1, Cons (expr2, ...Cons (exprn, Nil) ...))
```

Remark

- Lists are values (can be used in the language constructs and functions)
- Order matters

DEMO: some list of integers

Remark Similarly, one can define lists of booleans, floats, functions. . . but it is tedious

Typing

One new rule: All elements of the list should *be of the same type*

Previous typing rules applies to lists (with `if`...`then`...`else`, pattern matching, functions)

DEMO: Illustration of typing rules

Remark Later we will see:

- `type list_of_t = Nil | Cons of t * list_of_t`
  is actually the type `t list` in OCaml, for any type `t`
- more convenient notations

(because lists are pre-defined in OCaml) □

Back on pattern matching

Good news, it works for lists!

Pattern matching: an expression describing a computation performed according to the “shape” (i.e., the pattern) of the given expression

- The shape is described using a filter/pattern
- The pattern allows to filter and name/extract values

Several possible shapes/patterns with lists:

Expected shape	**Filter**
the empty list	`Nil`
the non-empty list	`Cons` (_, `l`), `Cons` (_, _), `Cons` (`e`, `l`), `Cons` (`e`,_)
(dealing with integer) the list with only one element: the integer 2	`Cons` (2,`Nil`)
(dealing with integer) the (non-empty) list where the first element is 1	`Cons`(1,_), `Cons` (1,`l`)
...	...

Remark Equivalent filters differ by the identifier they name in the associated expressions

Some simple functions on list

DEMO: Simple functions and their alternative implementations

`type intlist = Nil | Cons of int * intlist`

Example (Put an int as a singleton list - `putAsList`)

- Profile: `putAsList`: `int` → `intlist`
- Description/Semantics: `putAsList n` is the singleton list with one element which is `n`
- Examples: `putAsList n` = `Cons (n,Nil)`

Example (Head of a list - `head`)

- Profile: `head`: `intlist` → `int`
- Description/Semantics: `head l` is the first element of list `l`, and returns an error message if the list is empty
- Exs: `head (Cons (1,Nil))` = 1, `head Nil` = "`error message`", ...

Example (Other functions)

- `remainder`
- `is_zero_the_head`
- `second`

Dealing with empty lists

Four alternatives

1. return error message (with `failwith` command), as in the previous demo
2. define a specific type: the *non-empty lists*

```
type nonempty_intlist =
    Elt of int
   | Cons of int * nonempty_intlist
```

3. return a boolean with the result indicating whether it should be considered/ is meaningful
   ↪result usage is guarded by the returned boolean
4. not consider the empty list in the function:
   ↪ thus one accepts the warning provided by the pattern matching
   ↪ be careful when calling the function

DEMO: Four alternatives on the function `head`

Recursive functions on lists

Most problems on lists solved using recursion/induction because lists are a recursive type

A list is either

a) the empty list
b) a non-empty list

Remark Similarity with Peano numbers □

Body of a recursive function on lists

Consists in a case analysis "mimicking/following" the structure of the argument list

a) treatment for the empty list (Nil)
b) treatment for the non-empty list (Cons (elt,remainder)):

computation depending on 1) the current element 2) the result of the function on the remainder

↪ defining the function on cases **a)** and **b)** suffices to define the function

To define `f`: `list_of_t1` → `t2`, a recursive function:

a) `f Nil` = `...some value in t2...`
b) `f (Cons (elt, remainder))` = `g ( h elt, f remainder)`
where `g`: `t1` → `t3` and `g`: `t3` → `t2` → `t2`

Defining some recursive functions on lists

Example (Length of a list)

The length of a list is its number of elements

- Profile: `length`: `intlist` $\rightarrow$ `int`
- Semantics: `length l` = $|l|$, the number of elements
- Examples: `length Nil`=0, `length (Cons(9,Nil))`=1...
- Recursive equations:

$$\begin{aligned} \text{length} \quad \mathit{Nil} &= 0 \\ \text{length} \quad (\mathit{Cons}(a, l)) &= 1 + \text{length} \quad l \end{aligned}$$

- Termination:
  - Let's define $\text{measure}(\text{length l}) = \text{size}(\text{l})$ where $\text{size}(\text{l})$ is the number of applications of the constructor `Cons` to get l
  - We have: $\text{measure}(\text{length} \quad \text{Cons}(\_, \text{l})) > \text{measure}(\text{length l})$
- Implementation:

```
let rec length (l:intlist):int=
 match l with
  | Nil → 0
  | Cons (_,l) → 1+length l
```

DEMO: Example of execution of `Cons(1,Cons(2,Nil))`

Defining some recursive functions on lists - ctd

Lists of integers

Example (Lists of integers)

- `sum`: returns the sum of the elements of the list
- `belongsto`: indicates whether an element belongs to a list
- `last_element`: returns the last element of a list
- `minimum`: returns the minimum of a list of integers
- `interval`: returns the interval, as a list, given the left and right bound of the interval
- `evens`: getting the even integers of a list
- `replace`: replacing all occurrences of an element by another element
- `concatenate`: concatenating two lists
- `split`: split a list of pairs into a pair of lists
- `is_increasing`: is a list in increasing order

Defining some recursive functions on lists - ctd

List of cards

Example (Lists of cards)

```
type card = Value of int | Jack | Queen | King | Ace
type hand = Nil | Cons of card * hand
```

- `value_card`: `card` → `int`
- `value_main`: `main` → `int`

OCaml pre-defined implementation of lists

OCaml proposes a pre-defined implementation of lists
(in the Standard library)

- `Nil` is noted [ ]
- `Cons` is replaced by the infix operator ::

Example (List in OCaml notation)

- `Cons` (2, `Nil`) is noted [2]
- `Cons` (4,`Cons` (9, `Nil`)) is noted 4::(9::[ ])

Some shortcuts (syntactic sugar):

- `v1`::(`v2`::...::(`vn`::[ ])) can be noted `v1`::`v2`:: ...`vn`::[ ]
- `v1`::`v2`::... `vn`::[ ] can be noted [`v1`;`v2`;...;`vn`]

Type: `list_of_t` becomes `t list`

DEMO: OCaml pre-defined lists

Back to the language constructs

Nothing changes

Same rules apply for `if`...`then`...`else` construct and function calls

Pattern matching: same rule/possibilities, different syntax:

Expected shape	**Filter**
the empty list	[]
the non-empty list	_::_ _::l e::_ e::l
(dealing with integer) the list with only one element: the integer 2	[2], 2::[]
(dealing with integer) the (non-empty) list where the first element is 1	1::l, 1::_
...	...

Revisiting the previous functions using OCaml predefined lists

Example (Lists of integers)

- `putAsList`, `head`, `remainder`, `is_zero_the_head`, `second`
- `sum`: returns the sum of the elements of the list
- `belongsto`: indicates whether an element belongs to a list
- `last_element`: returns the last element of a list
- `minimum`: returns the minimum of a list of integers
- `interval`: returns the interval, as a list, given the left and right bound of the interval
- `evens`: getting the even integers of a list
- `replace1`: replacing all occurrences of an element by another element
- `concatenate`: concatenate two lists
- `is_increasing`: determines if a list is in increasing order
- `reverse`: produces the list as if the initial list is read from right to left

DEMO: Implementing some of these functions

Some functions using OCaml predefined lists

Example (`sublist`: is a list is a sublist of another?)

Indicates whether a list is a sublist of another by erasing
For example:

- [`e2` ; `e4` ; `e5`] is a subsequence of [`e1` ; `e2` ; `e3` ; `e4` ; `e5` ; `e6`]
- [`e2` ; `e4` ; `e5` ; `e7`] is NOT a subsequence of `e1` ; `e2` ; `e3` ; `e4` ; `e5` ; `e6`]
- [`e4` ; `e2` ; `e5`] is NOT a sublist of [`e1` ; `e2` ; `e3` ; `e4` ; `e5` ; `e6`]

Analysis:

- predicate taking two sequences as parameters
- the second sequence is obtained by erasing: elements of the first sequence are elements of the second sequence

DEMO: Implementing `sublist`

Example (Lists of integers)

- `zip`: takes a pair of lists and returns the list of corresponding pairs
-

DEMO: Implementing some of these functions

Some predefined functions in the list module

Functions (as we defined them)	OCaml implem
`nth`	`List.nth`
`length`	`List.length`
`head`	`List.hd`
`tail`	`List.tl`
`concatenate`	`@`, `List.append`
`reverse`	`List.rev`


List
http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

Previous Up Next

Module List

`module List: sig .. end`

List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

`val length : 'a list -> int`
Return the length (number of elements) of the given list.

`val hd : 'a list -> 'a`
Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

`val tl : 'a list -> 'a list`
Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

`val nth : 'a list -> int -> 'a`
Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.

`val rev : 'a list -> 'a list`
List reversal.

`val append : 'a list -> 'a list -> 'a list`
Catenate two lists. Same function as the infix operator `@`. Not tail-recursive (length of the first argument). The `@` operator is not tail-recursive either.

`val rev_append : 'a list -> 'a list -> 'a list`
`List.rev_append l1 l2` reverses `l1` and concatenates it to `l2`. This is equivalent to `List.rev l1 @ l2`, but `rev_append` is tail-recursive and more efficient.

`val concat : 'a list list -> 'a list`
Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result. Not tail-recursive (length of the argument + length of the longest sub-list).

`val flatten : 'a list list -> 'a list`
Same as `concat`. Not tail-recursive (length of the argument + length of the longest sub-list).

Iterators

`val iter : ('a -> unit) -> 'a list -> unit`
`List.iter f [a1; ...; an]` applies function `f` in turn to `a1; ...; an`. It is equivalent to `begin f a1; f a2; ...; f an; () end`.

`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 rev_map : ('a -> 'b) -> 'a list -> 'b list`
`List.rev_map f l` gives the same result as `List.rev (List.map f l)`, but is tail-recursive and more efficient.

`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.

Iterators on two lists

Sorting lists
Motivations

Sorting $\approx$ organizing a list according to some order (e.g., $<$ for `int`):

$$\text{unsorted list} \overset{\text{sorting}}{\longrightarrow} \text{sorted list}$$

Example

[2;1;9;4] $\overset{\text{sorting}}{\longrightarrow}$ [1;2;4;9]

- `type person = Toto | Titi | Tata`
- `[Titi;Tata;Toto]` $\overset{\text{sorting}}{\longrightarrow}$ `[Toto;Titi;Tata]`

Motivations?

- more informative, depending on the context
- easier to browse/modify
- ...

Several sorting algorithms that differ by

- how "fast" they are
- how "much memory" they need
- how they behave depending on the input (unsorted) list

→ "tasting some sorting algorithms"

Example (Searching an element in a sorted list)

It narrows the search (when one passes over the searched element)

```
let rec belongstosortedlist (e:int) (l:int list):bool=
 match l with
  | [] → false
  | x::lp → e=x || (e > x) && belongstosortedlist e lp
```

Example (Inserting an element in a sorted list)

```
let rec insert (e:int) (l:int list):int list=
 match l with
  | [] → [e]
  | x::lp → if e<x then e::l else x::(insert e lp)
```

Some sorting algorithms

to be implemented

Exercise: Sorting by insertion

"Isolate an element (e.g., the head), sort other elements, and then insert the isolated element at the correct position"

Exercise: sorting by selection

"Extract the least element which becomes the next on the resulting list"
Hints: you are going to need two functions:

- `min_list`: returns the minimal element of a list
- `suppress`: suppresses the first occurrence of an element in a list

Conclusion

Lists: a very practical data type

- Can be defined explicitly as a recursive union type
  - operators `Cons`, `Nil`
  - first-class citizens
  - typing rules apply
  - less practical: a lot to write, operators for each type of list
- We can use the syntactic sugar of OCaml: ::, [ ], `@`, [`v1`;`v2`;...;`vn`]
- Recursive functions on lists:
  - define the base case(s)
  - define the inductive case
- Sorting lists: insertion sort, selection sort

Assignment

- Double-check that you are able to **fully** define the functions of this lecture
- Revisit all functions that fail on some argument list and implement the alternatives, as seen for the `head` function
- Revisit all functions implemented "à la Lisp" using the shorter notation provided by OCaml
- Visit OCaml standard library on List (find the implemented functions in the lecture + play/test the other functions)