Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 3 - Sequence 1: Recursive types

Deep data structures

- Some standard data structures like lists and trees have an **unbounded depth**.
- We cannot define a type for lists because we have only seen "flat" data types.
- Informally, a list of integers is **either**:
  - an empty list, or
  - an integer and the *rest of the list*.
- We already know how to define a type by cases using sum types.
- Now, just realize that the "rest of the list" is also a list.

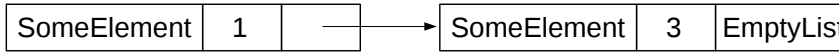
The type for list of integers I

```
type int_list =
  | EmptyList
  | SomeElement of int * int_list;;
# type int_list = EmptyList | SomeElement of int * int_list
```

In the machine

- The following value:
  `SomeElement (1, SomeElement (3, EmptyList));;`

... implements a **linked list** data structure:



Recursive types

- A sum type can refer to itself in its own definition.
- Such a sum type is therefore **recursive**.
- Functions over a recursive type are often defined by case analysis and recursion.

Computing the length of a list I

```
let rec length = function
  | EmptyList -> 0
  | SomeElement (x, l) -> 1 + length l;;
# val length : int_list -> int = <fun>
```

A predefined type for lists

- The type for lists of elements of type `t` is predefined in *OCaml* and written:

```
t list
```

- The empty list is written:

```
[]
```

- `[]` is a special tag corresponding to `EmptyList` in the previous example.
- An integer `i` followed by the rest of the list `r` is written:

```
i :: r
```

- `::` is a special tag corresponding to `SomeElement`.
- A list can be defined by enumeration:

```
[ some_expression; ...; some_expression ]
```

Computing the length of a *OCaml* list I

```
let rec length = function
  | [] -> 0
  | x :: xs -> 1 + length xs;;
# val length : 'a list -> int = <fun>
let three = length [1; 2; 3];;
# val three : int = 3
```

Reversing a list in quadratic time I

```
(* The '@' is a predefined operator that appends a list to another one. *)
let rec rev = function
  | [] -> []
  | x :: xs -> rev xs @ [ x ];;
# val rev : 'a list -> 'a list = <fun>
let l = rev [ 1; 2; 3 ];;
# val l : int list = [3; 2; 1]
```

Reversing a list in linear time I

```
let rec rev_aux accu = function
  | [] -> accu
  | x :: xs -> rev_aux (x :: accu) xs;;
# val rev_aux : 'a list -> 'a list -> 'a list = <fun>
let rev l = rev_aux [] l;;
# val rev : 'a list -> 'a list = <fun>
let l = rev [1; 2; 3];;
# val l : int list = [3; 2; 1]
```

Remove repeated elements I

```
let rec uniq = function
  | [] -> []
  | [x] -> [x]
  | x :: x' :: xs ->
    if x = x' then
      uniq (x' :: xs)
    else
      x :: uniq (x' :: xs);;
# val uniq : 'a list -> 'a list = <fun>
let l1 = uniq [1;2;2;3;4;3];;
# val l1 : int list = [1; 2; 3; 4; 3]
```