Exam of Programming Languages

8 July 2016

Disclaimer. Note that to have a running solution for an exercise is not enough: you need a well-cooked solution that proves your ability to use what explained during the classes. All the exercises have the same value: 11; Exercise are valued separately and it is necessary to get at least 6 points each exercise to pass the exams, i.e., 18 achieved with only 2 exercises means to fail the exam.

Exercise ML/OCaML: A «Continued» QuickSort.

As you probably remember from the lectures, functional programming permits to write a really elegant implementation for the quicksort algorithm:

```
let qsort (>:) l =
  let rec qsort = function
    [] -> []
  | h::tl -> (qsort (List.filter (fun x -> (x >: h)) tl) )
             @ [h] @
             (qsort (List.filter (fun x -> (h >: x)) tl) )
  in qsort l
```

Unfortunately, this implementation is not efficient as it could be and the call stack could explode with very large lists. Mainly this is due to the fact that it is not tail recursive. The quickstort algorithm can't be easily transformed into a recursive version. The only way is to use the *continuation passing style*.

For those not remembering the lectures (look at the one on functors), a continuation reifies the program control state, i.e., the continuation is a data structure that represents the computational process at a given point in the process's execution; the created data structure can be accessed by the programming language, instead of being hidden in the runtime environment. The term continuation is also be used to refer to first-class continuations, which are constructs that give a programming language the ability to save the execution state at any point and return to that point at a later point in the program, possibly multiple times. This is illustrated by the *continuation sandwich* description:

> «Say you're in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it. :-)» [Palmer'04]

In this description, the sandwich is part of the program data (e.g., an object on the heap), and rather than calling a *make sandwich* routine and then returning, the person called a *make sandwich with current continuation* routine, which creates the sandwich and then continues where execution left off.

This exercise consists in using continuations in order to write a tail recursive implementation of the quicksort algorithm, named `cqsort`. Please note that there will be only one possible solution that can be considered corrected, that is, the one that use continuation-passing style.

This is a possible main, that will be used to test your assignments. Please note that `Qsort` is the module with the old `qsort` implementation, `Cqsort` is the module with your assignment and the reported figures depends on the machine you are using.

```
open Qsort ;;
open Cqsort ;;

let make_list x =
   let rec make_list x acc =
      if x = 0 then acc
      else make_list (x-1) (x::acc)
   in make_list x [];;

let profile f =
   let s = Sys.time() in
      let _ = f() in (Sys.time()) -. s;;

let main () =
  let l = make_list 10000 in
    Printf.printf
       "Sorting 10000 elements with the functional qsort takes %3.2f µs\n"
       (profile (fun () -> qsort (>) l));
    Printf.printf
```

```
        "Sorting 10000 elements with the continuation qsort takes %3.2f µs\n"
        (profile (fun () -> cqsort (>) l));;

let() = main() ;;
```

```
[16:16]cazzola@hymir:~/lp/ocaml> ./main
Sorting 10000 elements with the functional qsort takes 7.34 µs
Sorting 10000 elements with the continuation qsort takes 6.36 µs
```

Exercise Erlang: Joseph's Problem.

Flavius Josephus was a roman historian of Jewish origin. During the Jewish-Roman wars of the first century AD, he was in a cave with fellow soldiers, 40 men in all, surrounded by enemy Roman troops. They decided to commit suicide by standing in a ring and counting off each third man. Each man so designated was to commit suicide... Josephus, not wanting to die, managed to place himself in the position of the last survivor.

In the general version of the problem, there are n Hebrews numbered from 1 to n and each k-th Hebrew will be eliminated. The count starts from the first Hebrew. What is the number of the last survivor?

The exercise consists of implementing a solution for the general version of the «Joseph's Problem». The Hebrews are represented by actors connected in a ring and every time one Hebrew is eliminated the corresponding actor is removed from the ring. The count is done by letting a message circulate on the ring and after k hops the actor which receives it is the one appointed to commit suicide. After that the count restarts from the actor next to the one eliminated. When only one actor remains alive this communicate master process that he is the survivor.

To recap, two modules are needed, `hebrew` for the actors describing the Hebrews and `joseph` implementing the homonymous function `joseph` that creates the ring according to the n and k passed as arguments and coordinates the initial and final messages. The following is a possible test case to verify the correct behavior of your assignment (run it with `erl -noshell -eval 'test:test()' -s init stop`.)

```
-module(test).
-export([test/0]).

test() ->
  joseph:joseph(30,3),
  joseph:joseph(300,1001),
  joseph:joseph(3000,37),
  joseph:joseph(26212,2025),
  joseph:joseph(1000,1000),
  joseph:joseph(2345,26212),
  joseph:joseph(100000,7).
```

```
[0:37]cazzola@hymir:~/lp/erlang>erl -noshell -eval 'test:test()' -s init stop
In a circle of 30 people, killing number 3
Joseph is the Hebrew in position 29
In a circle of 300 people, killing number 1001
Joseph is the Hebrew in position 226
In a circle of 3000 people, killing number 37
Joseph is the Hebrew in position 1182
In a circle of 26212 people, killing number 2025
Joseph is the Hebrew in position 20593
In a circle of 1000 people, killing number 1000
Joseph is the Hebrew in position 609
In a circle of 2345 people, killing number 26212
Joseph is the Hebrew in position 1896
In a circle of 100000 people, killing number 7
Joseph is the Hebrew in position 27152
```

All the solutions not conform to the specification will be considered wrong.

Exercise Scala: WtF -- What the F...un?

WtF could be considered as the language interpreting the smallest algebra describing integer values. That is the only operators are `0` (that represents the zero) `+` and `-` representing the successor and the predecessor of a given number; therefore `0+++` is 3 and `0-` is -1 (note that operators are applied in postfix notation).

Apart from this, the language provides only conditionals (the `?:` operator, as in «cond» ? [«block$_1$»] : [«block$_2$»]) that matches a value against 0 to decide which branch to execute (the code in the branches is enclosed in square parenthesis that **cannot** be nested) and the possibility to define recursive functions. Last but not least, the `!` operator (postfixed) is used to print (ended by a carriage return) the value just calculated.

All functions are defined before the code to evaluate and each definition is introduced by the `def` statement and ends with the end of line. A function has a name (a capital letter; so there could be no more than 26 user-defined functions) and the number of arguments they must take when invoked; their invocation is postfixed. The body definition is separated from the function heading by the = symbol; the body can contain any expression admissible in the language included calls to other functions or to the function we are defining. In the body, and only in the body, you can refer to the actual parameters through their position (prefixed by the `$` symbol) in the call, e.g., `$1` is the first argument, `$2` the second and so on.

Note that to get the exercise correctly evaluated **you have to submit all the files** related to the exercise, main program included.

The following is an example of code with the expected execution

```
def H 1 = $1 !
def S 2 = $2 ? [$1] : [$1+$2-S]
def M 2 = $2 ? [$1] : [$1-$2-M]

def A 3 = $2 ? [$1] : [$1 $3 S $2- $3 A]
def P 2 = 0 $2 $1 A

def X 2 = $2 ? [$1] : [$1 $2 P $2- X]
def F 1 = $1 $1- X

0+++++++!

0 H
0+++ H

0- ? [0+!] : [0-!]

0++++++ 0+++ S!
0++++++ 0+++ M!
0+++ 0++++++ M!
0+ 0++++++++++ M 0++++S!
0++ 0++++ S 0++++ S!
0+++++ 0+++++++ P!
0++++ 0++ M 0+++++++ P!
0++ 0++++ P 0+++++ P!
0+++++F!
```

```
[9:33]cazzola@surtur:~/scala>scala WtFEvaluator ops.wtf
7
0
3
-1
9
3
-3
-5
10
35
14
40
120
```