Principles of Programming Languages, 2019.02.08

Notes
- Total available time: 1h 40'.
- You may use any written material you need, and write in Italian, if you prefer.
- You cannot use electronic devices during the exam.

Exercise 1, Scheme (10 pts)

Define a pure function *multi-merge* with a variable number of arguments (all of them must be ordered lists of numbers), that returns an ordered list of all the elements passed. It is forbidden to use external sort functions.

E.g. when called like:

(multi-merge '(1 2 3 4 8) '(-1 5 6 7) '(0 3 8) '(9 10 12))

it returns: *'(-1 0 1 2 3 3 4 5 6 7 8 8 9 10 12)*

Exercise 2, Haskell (14 pts)

We want to define a data structure, called *BFlist* (*Back/Forward list*), to define lists that can either be "forward" (like usual list, from left to right), or "backward", i.e. going from right to left.

We want to textually represent such lists with a plus or a minus before, to state their direction: e.g. +[1,2,3] is a forward list, -[1,2,3] is a backward list.

Concatenation (let us call it <++>) for BFlist has this behavior: if both lists have the same direction, the returned list is the usual concatenation. Otherwise, forward and backward elements of the two lists delete each other, without considering their stored values.

For instance: *+[a,b,c]* <++> *-[d,e]* is *+[c]*, and *-[a,b,c]* <++> *+[d,e]* is *-[c]*.

1) Define a datatype for BFlist.

2) Make BFList an instance of Eq and Show, having the representation presented above.

3) Define <++>, i.e. concatenation for BFList.

4) Make BFList an instance of Functor.

5) Make BFList an instance of Foldable.

6) Make BFList an instance of Applicative.

Exercise 3, Erlang (8 pts)

Consider the following code:

```
buffer(Data) ->
    case Data of
        empty ->
            receive
                {put, V} ->
                    buffer(V)
            end;
        _ ->
            receive
                {get, Who} ->
                    Who ! Data,
                    buffer(empty)
            end
    end.

create_buffers(0) -> [];
create_buffers(V) -> [spawn(?MODULE, buffer,
[empty]) | create_buffers(V-1)].
```

```
pi(_, []) -> ok;
pi(Buf, [{V,D} | Cmds]) ->
    lists:nth(V,Buf) ! {put, D},
    pi(Buf, Cmds).

po([]) -> ok;
po([V | Vs]) ->
    V ! {get, self()},
    receive
        X -> true
    end,
    io:format("~s~n", [X]),
    po(Vs).

run(Commands) ->
    Buf = create_buffers(length(Commands)),
    spawn(?MODULE, pi, [Buf, Commands]),
    spawn(?MODULE, po, [Buf]),
    ok.
```

1) Describe what this system does and how it works – a high level description is required, not a line-by-line commentary. What should be the output of the following call?

run([{3,house},{1,moose},{2,lambda},{4,dark}]).

2) Is it possible to have deadlocks in this system? Explain why.

3) Consider a successful run of the system: at the end, are all the processes done, or it is possible to have some of them still running? Explain why and how to fix this, if needed.

Solutions

Es 1

```
(define (merge a1 a2)
  (cond ((null? a1) a2)
        ((null? a2) a1)
        (else (let ((x (car a1))
                    (y (car a2)))
                (if (< x y)
                    (cons x (merge (cdr a1) a2))
                    (cons y (merge a1 (cdr a2))))))))

(define (multi-merge . data)
  (foldl merge '() data))
```

Es 2

```
data Dir = Fwd | Bwd deriving Eq
data BFlist a = BFlist Dir [a] deriving Eq

instance Show Dir where
  show Fwd = "+"
  show Bwd = "-"

instance (Show a) => Show (BFlist a) where
  show (BFlist x y) = show x ++ show y

instance Functor BFlist where
  fmap f (BFlist d x) = BFlist d (fmap f x)

instance Foldable BFlist where
  foldr f i (BFlist d x) = foldr f i x

(BFlist _ []) <++> x = x
x <++> (BFlist _ []) = x
(BFlist d1 x) <++> (BFlist d2 y) | d1 == d2 = BFlist d1 (x ++ y)
(BFlist d1 (x:xs)) <++> (BFlist d2 (y:ys)) = (BFlist d1 xs) <++> (BFlist d2 ys)

bflconcat (BFlist d v) = foldr (<++>) (BFlist d []) (BFlist d v)
bflconcatmap f x = bflconcat $ fmap f x

instance Applicative BFlist where
  pure x = BFlist Fwd [x]
  x <*> y = bflconcatmap (\f -> fmap f y) x
```

Es 3

Description:
It is a producer (PI)/consumer (PO) system with multiple buffers, where each buffer can contain only one datum. The producer performs a sequence of push operations {position of buffer, datum}, then ends. The consumer reads the content of the buffers in a sequence starting from the first and ending on the last.

The output should be:
moose
lambda
house
dark

Of course, deadlock is possible, e.g. if in the sequence of push there are two positions that are equal, and PI's first push happen before PO's get. E.g. [{3,house},{3,moose}]

Cleanup: if everything goes well, PI, PO, and the parent process end, but all the buffers are still running. An easy way to avoid this is to replace *buffer(empty)* in the buffer code with *ok*.