Principles of Programming Languages, 2017.09.01

Notes

- Total available time: 1h 30'.
- 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)

Consider a list L of symbols. We want to check if in L there are matching "a" and "b" symbols or "1" and "2" symbols, where "a" and "1" have an open parenthesis role, while "b" and "2" stand for close parenthesis respectively (i.e. a Dyck language); other symbols are ignored. Define a pure and tail recursive function *check-this* which returns the number of matching pairs, and #f if the parenthesis structure is not respected.

E.g.

(check-this '(a b a b)) is 2

(check-this '(h e l l o)) is 0

(check-this '(6 h a b a 1 h h i 2 b z)) is 3

(check-this '(6 h a b a 1 h h i b z 2)) is #f (wrong structure)

Exercise 2, Haskell (12 pts)

1) Define an infinite binary tree data structure, called *Itree*, i.e. a binary tree containing an infinite number of nodes, each containing a value, and in which every path from the root downwards is infinite.

2) Is it possible to automatically derive Show for Itree? If yes, do it; if not, state why and write a reasonable implementation of *show* for Itrees.

3) Define *costItree* which takes a value x and returns an Itree where all the contained values are equal to x.

4) Define *list2Itree*, which takes an infinite list L and returns an Itree T in which every path from the root downwards contains the same sequence of values of those in L (of course, all values in L but the first are duplicated in T).

5) Make Itree an instance of Functor.

6) Define a function *takeLevels* which takes a natural number n and an Itree T and returns a (finite) binary tree that represents the top subtree of T from its root to level n. (Note that the root has level 0, its sons level 1, and so on.)

7) Define a function *applyAtLevel* which takes a function f, a predicate p on levels (i.e. a function Int → Bool), and an Itree T, and returns an Itree that is identical to T, but for levels that satisfy p: those are updated by applying f to their values.

E.g. *applyAtLevel (+1) odd $ constItree 1* is an Itree where the root has value 1, its sons 2, their sons 1, and so on, alternatively.

Note: you are required to write all the types of the functions you define.

Exercise 3, Erlang (10 pts)

Define a *parfind* (parallel find) operation, which takes a list of lists L and a value x, and parallely looks for x in every list of L – the idea is to launch one process for each list, searching for x. If x is found, parfind returns one of the lists containing x; otherwise, it returns false.

E.g. parfind([[1,2,3],[4,5,6],[4,5,9,10]], 4) could return either [4,5,6] or [4,5,9,10]; parfind([[1,2,3],[4,5,6],[4,5,9,10]], 7) is false.

Solutions

Es 1

```
(define (check-this ls)
  (define (check-par in stack num)
    (if (null? in)
      num
      (let ((x  (car in))
            (xs (cdr in)))
        (cond
          ((eq? x 'a)
           (check-par xs (cons 'b stack) num))
          ((eq? x '1)
           (check-par xs (cons '2 stack) num))
          ((member x '(2 b))
           (if (and (cons? stack)
                    (eq? x (car stack)))
             (check-par xs (cdr stack) (+ num 1))
             #f))
          (else
            (check-par xs stack num))))))
  (check-par ls '() 0))
```

Es 2

```
data Itree a = Node (Itree a) a (Itree a)

instance (Show a) => Show (Itree a) where
   show (Node l v r) = "Node (... " ++ show v ++ " ...)"

list2Itree :: [a] -> Itree a
list2Itree (x1:xs) = Node (list2Itree xs) x1 (list2Itree xs)

constItree v = list2Itree [v, v ..]

instance Functor Itree where
   fmap f (Node l v r) = Node (fmap f l) (f v) (fmap f r)

data Tree a = Leaf a | Branch (Tree a) a (Tree a) deriving Show
takeLevels :: Int -> Itree a -> Tree a
takeLevels 0 (Node _ v _) = Leaf v
takeLevels k (Node l v r) = Branch (takeLevels (k-1) l) v (takeLevels (k-1) r)

applyAtLevel :: (a -> a) -> (Int -> Bool) -> Itree a -> Itree a
applyAtLevel f pred tree = applyAtLevel' f pred tree 0
                      where applyAtLevel' f pred (Node l v r) level = Node
                            (applyAtLevel' f pred l (level+1))
                            (if pred level then f v else v)
                            (applyAtLevel' f pred r (level+1))
```

Es 3

```
worker(Dad, List, X) ->
   case lists:member(X, List) of
      true ->  Dad ! {found, List};
      false -> Dad ! nay
   end.

get_result(0) -> false;
get_result(V) ->
   receive
      nay -> get_result(V-1);
      {found, L} -> L
   end.

parfind(LofL, X) ->
   lists:foreach(fun(L) ->
                     spawn(?MODULE, worker, [self(), L, X])
               end,
               LofL),
   get_result(length(LofL)).
```