Principles of Programming Languages, 2018.02.05

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, Haskell (12 pts)

1) Define a *Graph* data-type, for directed graphs. Nodes hold some generic data, while edges have no data associated.

2) Define a *graph_lookup* function, to get the data associated with a node in the graph (or nothing if the node is not present).

3) Define an *adjacents* function, to check if two nodes are adjacent or not.

4) Make *Graph* an instance of Functor.

Exercise 2, Erlang (10 pts)

The *fixed-point* of a function f and a starting value x is the value $v = f^k(x)$, with $k > 0$, such that $f^k(x) = f^{k+1}(x)$. We want to implement a fixed-point code using two communicating actors:

1) Define the function for an *applier* actor, which has a state *S*, holding a value, and receives a function *f* from other actors: if *S = f(S)*, it sends back the result *S* and ends it computation; otherwise sends back a message to state that the condition *S = f(S)* has not been reached.

2) Define a function called *fix*, which takes as input a function and a starting value, and creates and uses an applier actor to implement the fixed-point.

Exercise 3, Scheme (10 pts)

Consider the following program, containing two errors:

```
(define (ap state equality)
  (let ((local state))
    (lambda (f)
      (let  ((new  (f local))
             (flag (equality new local)))
        (when flag
          (set! local new))
        (cons flag new)))))
(define (g f v equality)
  (let ((alpha (ap v equality)))
    (let beta ((v (alpha f)))
      (call/cc
       (lambda (done)
         (when (car v)
           (done (cdr v)))
         (beta (alpha f)))))))
```

1) Describe how it works; 2) how can we rewrite *g* to avoid using *call/cc*; 3) why it is broken and how to fix it.

Solutions

Es 1

```
data Node a b = Node {
                  id :: a,
                  datum :: b,
                  adjacent :: [a]
                  } deriving Show
data Graph a b = Graph [Node a b] deriving Show  -- a more efficient version should be based on Data.Array

graph_lookup :: (Eq a) => Graph a b -> a -> Maybe (Node a b)
graph_lookup (Graph []) id = Nothing
graph_lookup (Graph ((Node i d a):xs)) id = if i == id then Just (Node i d a) else graph_lookup (Graph xs) id

adjacents :: Eq a => Graph a b -> a -> a -> Bool
adjacents g i j = case graph_lookup g i of
                    Nothing -> False
                    Just (Node _ _ adj) -> j `elem` adj

instance Functor (Node a) where
    fmap f (Node i d a) = Node i (f d) a

instance Functor (Graph a) where
    fmap f (Graph nodes) = Graph $ fmap (\x -> fmap f x) nodes
```

Es 2

```
applier(State) ->
    receive
        {Sender, F} -> NewState = F(State),
                      if
                          NewState =:= State -> Sender!{self(), State};
                          true -> Sender!{self(), no}, applier(NewState)
                      end
    end.

loop(P, F) ->
    P!{self(), F},
    receive
        {P, V} -> if
                      V =:= no -> loop(P, F);
                      true -> V
                  end
    end.

fix(F, V) ->
    A = spawn(?MODULE, applier, [V]),
    loop(A, F).
```

Es 3

It is a closure-based implementation of Exercise 2.

The *call/cc* construct can be easily changed in a (much clearer) *if*:

```
(define (g f v equality)
  (let ((alpha (ap v equality)))
    (let beta ((v (alpha f)))
      (if (car v)
          (cdr v)
          (beta (alpha f))))))
```

Errors: in the ap procedure, *when* should be *unless*, and the internal *let* should be a *let**.