Principles of Programming Languages

2013.09.05

Notes

- Total available time: 1h 30'.
- You may use any written material you need.
- You cannot use computers or phones during the exam.

1 Scheme (10 points)

Consider the *parent-pointer* implementation of general trees (PPT), where every node has local data (e.g. a name), and only one pointer, pointing to its parent. Of course, roots nodes do not point to other nodes.

PPTs are generally implemented as arrays containing pairs *(index of parent, node name)*, and one of such data structures can contain one or more different trees, e.g. using a Scheme-like syntax:
`#((? . R) (0 . A) (0 . B) (1 . C) (1 . D) (1 . E) (2 . F) (? . W) (7 . X) (7 . Y))`
where root nodes have first component "?". Nodes are usually referenced through their index.

PPTs are efficient for checking if two nodes belong to the same tree (we have just to check if the root is the same for both), so are often used to represent partitions.

You are requested to implement a **mutable** version of PPTs in Scheme. In particular, you must:

1. Define the operation *find-root*, to obtain the root of the tree containing the given node.
2. Define the operation *union!*, that takes two nodes and, if they belong to different trees, merges the two trees by making the root of the first node's tree the parent of the root of the second node's tree.

2 Haskell (12 points)

Define the data structure, *findRoot* and *union* operations for **immutable** PPTs in Haskell.
Remainder and hints: immutable arrays are offered by the module `Data.Array`; an array with index type `Int` and containing elements of type `Type` has type `Array Int Type`. You can use `Int` for indexes.

Typical operations on arrays are `!` to access an element (e.g. `A ! 3` is equivalent to `A[3]` in C), and `//` for updates (e.g. `A // [(3, 12)]` creates a new array containing the same elements of `A` but for `A[3] = 12`).

3 Prolog (9 points)

Define a `mixUp` predicate, which takes an arithmetic expression without variables containing only the * and + operators, and puts in its second argument the value of the expression obtained by changing all the * operators to + and vice versa, but keeping the same syntactic structure.
E.g. `mixUp(3*2+4,X)` must return `X = 20`.
Note: use cut to avoid unnecessary backtracks.

Solutions

Scheme

```
(define (parent tree node-index)
  (car (vector-ref tree node-index)))

(define (parent-set! tree node-index new-root)
  (vector-set! tree node-index
               (cons new-root
                     (node-name tree node-index))))

(define (node-name tree node-index)
  (cdr (vector-ref tree node-index)))

(define (find-root tree node-index)
  (let ((p (parent tree node-index)))
    (if p
        (find-root tree p)
        node-index)))

(define (union! tree node1 node2)
  (let ((r1 (find-root tree node1))
        (r2 (find-root tree node2)))
    (unless (= r1 r2)
      (parent-set! tree r2 r1))))

;; an example
(define my-tree (vector
                 '(#f . R)
                 '(0 . A)
                 '(0 . B)
                 '(1 . C)
                 '(1 . D)
                 '(1 . E)
                 '(2 . F)
                 '(#f . W)
                 '(7 . X)
                 '(7 . Y)
                 '(7 . Z)))
```

Haskell

```
import Data.Array

data Node = Parent Int | Root deriving (Show, Eq)
type ParentTree a = Array Int (Node, a)


parent :: ParentTree a -> Int -> Node
parent tree node = fst $ tree ! node

nodeName :: ParentTree a -> Int -> a
nodeName tree node = snd $ tree ! node
```

```
parentSet :: ParentTree a -> Int -> Int -> ParentTree a
parentSet tree node newRoot = tree // [(node, (Parent newRoot, nodeName tree node))]

findRoot tree node = case parent tree node of
  Root -> node
  Parent p -> findRoot tree p

union tree node1 node2 = let r1 = findRoot tree node1
                             r2 = findRoot tree node2
                         in if r1 /= r2
                            then parentSet tree r2 r1
                            else tree

-- an example tree:
myTree :: ParentTree String
myTree = listArray (0,10) [ (Root, "R"),
                            (Parent 0, "A"),
                            (Parent 0, "B"),
                            (Parent 1, "C"),
                            (Parent 1, "D"),
                            (Parent 1, "E"),
                            (Parent 2, "F"),
                            (Root    , "W"),
                            (Parent 7, "X"),
                            (Parent 7, "Y"),
                            (Parent 7, "Z")
                            ]
```

Prolog

```
mixUp(V,V) :- atomic(V), !.
mixUp(A1+A2,R) :- !,
        mixUp(A1,R1),
        mixUp(A2,R2),
        R is R1*R2.
mixUp(A1*A2,R) :- !,
        mixUp(A1,R1),
        mixUp(A2,R2),
        R is R1+R2.
```