Paradigme de programare

Haskell

Mihai Nan

Facultatea de Automatica si Calculatoare
Universitatea Politehnica din Bucuresti

Anul universitar 2020-2021

Grafuri

- Următoarele exerciții vor avea drept scop implementarea unei mici biblioteci pentru grafuri ORIENTATE.
- După cum știți, există mai multe modalități de reprezentare a unui graf. Biblioteca noastră va defini mai multe astfel de reprezentări, precum și algoritmi care operează pe grafuri.
- Algoritmii vor fi independenți de reprezentarea internă a grafului - ei vor funcționa indiferent de ce structură de date am ales noi pentru un anume graf.
- Pentru a obține această genericitate vom abstractiza noțiunea de graf într-o clasă care va expune operațiile pe care orice structură de date de tip graf ar trebui să le aibă.

Grafuri – Reprezentări

```haskell
-- reprezentam nodurile ca intregi
type Node = Int

-- reprezentam arcele ca perechi de noduri
type Arc = (Node, Node)
```

- Clasa Graph definește interfața pentru toate structurile de grafuri pe care le vom implementa mai jos.

Clasa Graph

```
class Graph g where
    -- Construieste un graf plecand de la o lista de noduri si arcele dintre
     ↪  noduri
    build :: [Node] -> [Arc] -> g
    -- Lista tuturor nodurilor din graf
    nodes :: g -> [Node] -- lista nodurilor din graf
    -- Lista arcelor din graf
    arcs :: g -> [Arc] -- lista muchiilor din graf
    -- Lista nodurilor catre care nodul dat ca parametru are un arc
    nodeOut :: g -> Node -> [Node]
    -- Lista nodurilor care au un arc catre nodul dat ca parametru
    nodeIn :: g -> Node -> [Node]
    -- Verifica daca exista un arc intre doua noduri
    arcExists :: g -> Node -> Node -> Bool
    -- Primeste un graf orientat si il transforma intr-un graf neorientat
    convertToUndirected :: g -> g
    -- Primeste un graf neorientat si il transforma intr-un graf orientat
    -- Se pastreaza doar muchiile de forma (u, v) cu proprietatea u < v
    convertToDirected :: g -> g
```

Clasa Graph – implementări implicite

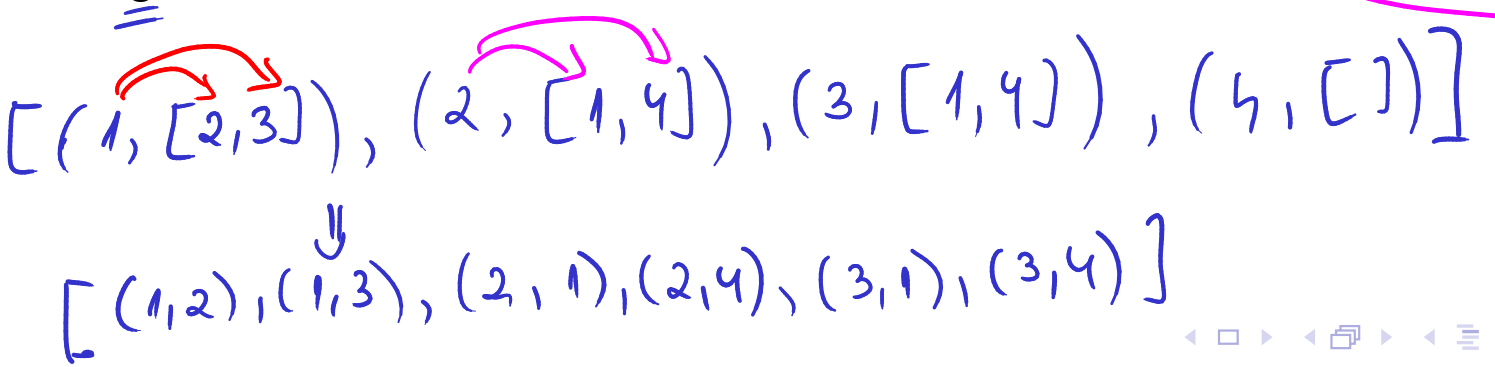
```
class Graph g where
  build :: [Node] -> [Arc] -> g
  nodes :: g -> [Node]
  arcs :: g -> [Arc]
  nodeOut :: g -> Node -> [Node]
  nodeIn :: g -> Node -> [Node]
  arcExists :: g -> Node -> Node -> Bool
  convertToUndirected :: g -> g
  convertToDirected :: g -> g
  -- implementari implicite
  arcExists g a b = (a, b) `elem` arcs g
  arcs g = [(i, j) | i <- ns, j <- ns, arcExists g i j] where ns = nodes g
  nodeIn g n = [x | x <- nodes g, arcExists g x n]
  nodeOut g n = [x | x <- nodes g, arcExists g n x]
  convertToUndirected g = build (nodes g) ((arcs g) ++ [(v, u) | (u, v) <- (arcs
   ↪   g)])
  convertToDirected g = build (nodes g) newArcs
    where
      newArcs = foldl (\acc (u, v) ->
                if (notElem (u, v) acc) && (notElem (v, u) acc && u < v) then
                  (u, v):acc
                else
                  acc) [] (arcs g)
```

Tipul AdjListGraph

- Definiți tipul AdjListGraph care reprezintă un graf ca pe o serie de perechi (nod, listă vecini). Includeți AdjListGraph în clasa Graph.

```
newtype AdjListGraph = ALGraph [(Node, [Node])] deriving Show

instance Graph AdjListGraph where
  build ns arcs = ALGraph [(node, adjacentNodes node) | node <- ns]
    where adjacentNodes node = [neighbour | (n, neighbour) <- arcs, n == node]
  nodes (ALGraph ps) = map fst ps
  arcs  (ALGraph g) = concatMap (\(node,neighL) -> map (\x -> (node, x)) neighL)
   ↪   g
```



Tipul ArcGraph

- Definiți tipul ArcGraph care reprezintă un graf ca o listă de noduri și o listă de arce între noduri. Includeți ArcGraph în clasa Graph.

```
data ArcGraph = AGraph [Node] [Arc] deriving (Show)

instance Graph ArcGraph where
        build nodes arcs = AGraph nodes arcs
        nodes (AGraph nodes _) = nodes
        arcs (AGraph _ arcs) = arcs
```

Funcția convert

g1 :: ArcGraph

g2 :: AdjListGraph

g2 = convert g1

```
convert :: (Graph g1, Graph g2) => g1 -> g2
convert g1 = build (nodes g1) (arcs g1)
```

g3 = convert g1
↳ eroare

Polimorfism

1. **Parametric**: manifestarea aceluiași comportament pentru parametri de tipuri diferite.
2. **Ad-hoc**: manifestarea unor comportamente diferite pentru parametri de tipuri diferite.

Parcurgerea grafurilor

- O traversare a unui graf este o listă de perechi `(nod, Maybe nodPărinte)`. Această structură va conține toate nodurile rezultate în urma unei parcurgeri a unui graf, în ordinea apariției în parcurgere și împreuna cu părintele nodului din parcurgere (Pentru un nod N, părintele său este nodul din care s-a ajuns la N în decursul parcurgerii).

```
-- O traversare a unui graf este o lista de perechi
type Traversal = [(Node, Maybe Node)]
-- O cale in graf este reprezentata ca lista de noduri
type Path = [Node]
-- Definitie pentru algoritmi de parcurgere a unui graf
type TraverseAlgo g = g -> Node -> Traversal
```

Observație

Tipul grafului este o variabilă de tip – algoritmii trebuie să funcționeze pentru orice structură de tip graf.

Parcurgerea grafurilor

- DFS și BFS descriu în principiu același proces - folosesc o structură de date în care sunt ținute nodurile care trebuie parcurse în continuare, precum și o structură cu nodurile vizitate deja.
- Ceea ce diferă este structura de date (DFS - stiva, BFS - coada).
- Vom folosi o listă pentru a reprezenta ambele structuri de date - și deci ce variază este modul în care adaugăm nodurile de parcurs - la DFS adăugăm la începutul listei, la BFS la sfârșit.

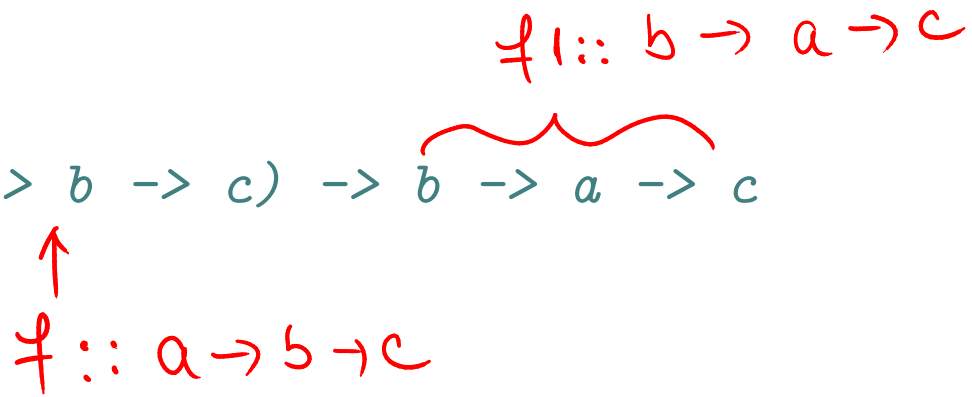
```
mkTraversal accumOp g start = iter [(start, Nothing)] []
  where
    iter [] visited = reverse visited
    iter (p@(x, parent):xs) visited
      | any ((==x).fst) visited  = iter xs visited
      | otherwise =
        let additions = [(y, Just x) | y <- nodeOut g x]
        in  iter (accumOp additions xs) (p:visited)
```

Parcurgerea grafurilor

```haskell
-- Algoritmul de parcurgere in adancime (Depth-First Search)
dfs :: Graph g => TraverseAlgo g
dfs = mkTraversal (++)

-- Algoritmul de parcurgere in latime (Breadth-First Search)
bfs :: Graph g => TraverseAlgo g
bfs = mkTraversal (flip (++))

-- > :t flip
-- flip :: (a -> b -> c) -> b -> a -> c
```



Sinteza de tip – I

```
f x y = (x y) y
```

$f :: a \to b \to c$

$x :: d \to e$, $x :: a$ $\Rightarrow a = d \to e$

$(x\ y) :: b \to c$, $(x\ y) :: e$ $\Rightarrow e = b \to c$

$\Rightarrow a = d \to b \to c$

$\boxed{d = b}$ pt că $x :: d \to e$ și x este apelat cu par. y (cărui este de tip b)

$\Rightarrow a = b \to b \to c$

$f :: (b \to b \to c) \to b \to c$

Sinteza de tip – II

```
f x y = x (f y x)
```

$f :: a \to b \to c$

⇓

tipul lui x = tipul lui y $\Rightarrow a = b$

x (f y x) $\Rightarrow$ x este o funcție ce primește ca parametru rezultatul lui (f y x) $\Rightarrow$ tipul c și returnează rezultatul lui f x y $\Rightarrow$

$\Rightarrow$ tipul c $\Rightarrow$ $x :: c \to c$

$f :: (c \to c) \to (c \to c) \to c$

```
[1,2,3] !! 1
```

Sinteza de tip – III