Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Ana Cristina Țurlea

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
ana.turlea@fmi.unibuc.ro

1. Declararea tipurilor cu **newtype**
2. Clase de tipuri
3. Semantica denotațională în Haskell
4. Mini-Haskell

Declararea tipurilor cu **newtype**

Declararea tipurilor cu **newtype**

- **newtype** se folosește când avem un singur constructor de date cu un singur argument
- folosind **newtype** facem o copie a unui tip de date deja existent
- spre deosebire de tipurile definite cu **type**, cele definite cu **newtype** pot deveni instanțe ale unor clase

```
type Name = String
newtype NewName = N String

instance Show Name where
  show (s) = "*"++ s ++ "*"
```

Illegal instance declaration for ‘Show Name’

```
instance Show NewName where
  show (N s) = "*"++ s ++ "*"

> N "abd"
*abd*
```

Exemplu: **newtype**

```
type Key = Int
type Value = String
newtype PairList = PairList{getPairList :: [(Key, Value)]}
```

- datele de tip PairList sunt definite folosind înregistrări
- o dată de tip PairList are una din formele
  - PairList [(k1, v1),...,(kn,vn)]
  - PairList { getPairList = [(k1,v1),...,(kn,vn)] }
- getPairList este funcția proiecție:

```
getPairList :: PairList -> [(Key,Value)]
```

De exemplu
getPairList (PairList [(1, "m1"),(2,"m2")])=[(1, "m1"),(2,"m2")]

Clase de tipuri

- Vom exersa manipularea listelor si tipurilor de date prin implementarea catorva colectii cu elemente de tip cheie-valoare.
- Aceste colectii vor trebui sa aiba urmatoarele facilități
  - crearea unei colectii vide
  - crearea unei colectii cu un element
  - adaugarea/actualizarea unui element intr-o colectie
  - cautarea unui element intr-o colectie
  - stergerea (marcarea ca sters a) unui element dintr-o colectie
  - obtinerea listei cheilor
  - obtinerea listei valorilor
  - obtinerea listei elementelor
  - conversii intre liste si colectii

    ...

O clasă de tipuri

```
type Key = Int
type Value = String

class Collection c where
  cempty :: c
  csingleton :: Key -> Value -> c
  cinsert :: Key -> Value -> c -> c
  cdelete :: Key -> c -> c
  clookup :: Key -> c -> Maybe Value
  ctoList :: c -> [(Key, Value)]
```

Definți operațiile derivate

```
  ckeys :: c -> [Key]
  cvalues :: c -> [Value]
  cfromList :: [(Key,Value)] -> c
```

O clasă de tipuri

```
class Collection c where
  cempty :: c
  csingleton :: Key ->  Value -> c
  cinsert:: Key -> Value -> c  -> c
  cdelete :: Key -> c  -> c
  clookup :: Key -> c -> Maybe Value
  ctoList :: c  -> [(Key, Value)]

  ckeys :: c  -> [Key]
  cvalues :: c  -> [Value]

  ckeys c = [fst p | p <- ctoList c]
  cvalues c = [snd p | p <- ctoList c]
```

O clasă de tipuri

```
class Collection c where
  cempty :: c
  csingleton :: Key ->  Value -> c
  cinsert:: Key -> Value -> c  -> c
  cdelete :: Key -> c  -> c
  clookup :: Key -> c -> Maybe Value
  ctoList :: c  -> [(Key, Value)]

  cfromList :: [(Key,Value)] -> c

  cfromList [] = cempty
  cfromList ((k,v):es)  = cinsert k v (cfromList es)
```

O clasă de tipuri

```
type Key = Int
type Value = String
class Collection c where
  cempty :: c
  csingleton :: Key -> Value -> c
  cinsert :: Key -> Value -> c -> c
  cdelete :: Key -> c -> c
  clookup :: Key -> c -> Maybe Value
  ctoList :: c -> [(Key, Value)]
  ckeys :: c -> [Key]
  cvalues :: c -> [Value]
  cfromList :: [(Key,Value)] -> c
-- minimum definition:
-- cempty, csingleton, cinsert, cdelete, clookup, ctoList
  ckeys c = [fst p | p <- ctoList c]
  cvalues c = [snd p | p <- ctoList c]
  cfromList [] = cempty
  cfromList ((k,v):es) = cinsert k v (cfromList es)
```

O clasă de tipuri

Fie tipul listelor de perechi de forma cheie-valoare:

```
newtype PairList
  = PairList { getPairList :: [(Key,Value)] }

instance Show PairList where
  show (PairList ps) = "PairList " ++ (show ps)
```

Faceti PairList instanta a clasei Collection.
Trebuie să definim
cempty, csingleton, cinsert, cdelete, clookup, ctoList

O clasă de tipuri

```
newtype PairList
  = PairList { getPairList :: [(Key,Value)] }

instance Collection PairList where
    cempty = PairList []
    csingleton k v = PairList [(k,v)]
    cinsert k v (PairList l) = PairList $ (k,v):filter ((/= k
        ). fst) l
    clookup k = lookup k . getPairList
    cdelete k (PairList l) = PairList $ filter ((/= k). fst)
        l
    ctoList = getPairList
```

O clasă de tipuri

Fie tipul arborilor binari de cautare (ne-echilibrati):

```
data SearchTree = Empty
                | Node
                    SearchTree      -- elem. chei mai mici
                    Key             -- cheia
                    (Maybe Value)   -- valoarea
                    SearchTree      -- elem. chei mai mari
                  deriving Show
    deriving Show
```

Observati ca tipul valorilor este ‘Maybe value‘. Acest lucru se face pentru a reduce timpul operatiei de stergere prin simpla marcare a unui nod ca fiind sters. Un nod sters va avea valoarea ‘Nothing‘.
Faceti ‘SearchTree‘ instanta a clasei ‘Collection‘.
Trebuie să definim
cempty, csingleton, cinsert, cdelete, clookup, ctoList

O clasă de tipuri

```
data SearchTree = Empty
                | Node
                    SearchTree      -- elem. cheie mai mici
                    Key             -- cheia
                    (Maybe Value)   -- valoarea
                    SearchTree      -- elem. cheie mai mari
                  deriving Show

instance Collection SearchTree where
    cempty = Empty
    csingleton k v = Node Empty k (Just v) Empty
    cinsert k v = ...
    cdelete k =  ...
    clookup k = ...
    ctoList t = ...
```

O clasă de tipuri

```
data SearchTree = Empty
                | Node
                    SearchTree      -- elem. chei mai mici
                    Key             -- cheia
                    (Maybe Value)   -- valoarea
                    SearchTree      -- elem. chei mai mari
                  deriving Show
```

```
cinsert k v = go
  where
    go Empty = csingleton k v
    go (Node t1 k1 v1 t2)
      | k == k1   = Node t1 k1 (Just v) t2
      | k < k1    = Node (go t1) k1 v1 t2
      | otherwise = Node t1 k1 v1 (go t2)
```

O clasă de tipuri

```
data SearchTree = Empty
                | Node
                    SearchTree      -- elem. chei mai mici
                    Key             -- cheia
                    (Maybe Value)   -- valoarea
                    SearchTree      -- elem. chei mai mari
                deriving Show
```

```
cdelete k = go
  where
    go Empty = Empty
    go (Node t1 k1 v1 t2)
      | k == k1    = Node t1 k1 Nothing t2
      | k < k1     = Node (go t1) k1 v1 t2
      | otherwise = Node t1 k1 v1 (go t2)
```

O clasă de tipuri

```
data SearchTree = Empty
                | Node
                    SearchTree      -- elem. chei mai mici
                    Key             -- cheia
                    (Maybe Value)   -- valoarea
                    SearchTree      -- elem. chei mai mari
                deriving Show
```

```
clookup k = go
  where
    go Empty = Nothing
    go (Node t1 k1 v1 t2)
       | k == k1    = v1
       | k < k1     = go t1
       | otherwise = go t2
```

O clasă de tipuri

```
data SearchTree = Empty
                | Node
                    SearchTree     -- elem. chei mai mici
                    Key            -- cheia
                    (Maybe Value)  -- valoarea
                    SearchTree     -- elem. chei mai mari
                deriving Show
```

```
ctoList Empty = []
ctoList (Node ltk k v gtk) = ctoList ltk ++ embed k v ++
    ctoList gtk
  where
    embed k (Just v) = [(k,v)]
    embed _ _ = []
```

O clasă de tipuri

```
class Collection c where
-- minimum definition: cempty, csingleton, cinsert, cdelete,
     clookup, ctoList
-- derived operations: ckeys, cvalues, cfromList

instance Collection PairList where ...

instance Collection SearchTree where ...


> cfromList [(1,"a"),(2,"b"),(3,"c")] :: PairList
PairList [(1,"a"),(2,"b"),(3,"c")]

> cfromList [(1,"a"),(2,"b"),(3,"c")] :: SearchTree
Node (Node (Node Empty 1 (Just "a") Empty) 2 (Just "b")
    Empty) 3 (Just "c") Empty
```

Semantica denotațională în Haskell

Feluri de a da semantica

- Limbaj de programare: sintaxă și semantică
- Feluri de semantică
  - Limbaj natural — descriere textuală a efectelor
  - Operațională — asocierea unei demonstrații a execuției
  - Axiomatică — Descrierea folosind logică a efectelor unei instrucțiuni
  - Denotațională — prin asocierea unui obiect matematic (denotație)

Limbajul unui mini calculator

Definim în Haskell limbajul unui mini calculator:

```
data Prog  = On Instr
data Instr  = Off | Expr :> Instr
data Expr  =  Mem | V Int | Expr :+ Expr
```

Semantica în limbaj natural

Dorim ca un program să afișeze lista valorilor corespunzătoare expresiilor, unde Mem reprezintă ultima valoare calculată. Valoarea inițială a lui Mem este 0.

De exemplu, programul

On ((V 3) :> ((Mem :+ (V 5)) :> Off))

va afișa lista [3,8]

Semantica denotațională - domenii semantice

Categoriilor sintactice le corespund domenii semantice

Categorii sintactice	Domenii semantice
Prog	$\mathbb{Z}^*$
Instr	$\mathbb{Z} \to \mathbb{Z}^*$
Expr	$\mathbb{Z} \to \mathbb{Z}$

unde $\mathbb{Z}$ este mulțimea numerelor întregi. Observăm că domeniile semantice pentru Instr și Expr sunt funcții deoarece depind de valoarea din memorie.

Domeniile semantice în Haskell

```
type Env = Int --valoarea celulei de memorie
type DomProg = [Int]
type DomInstr = Env -> [Int]
type DomExpr = Env -> Int
```

Semantica denotațională

Pentru a defini semantica denotațională trebuie să evaluăm (interpretăm) categoriile sintactice în domeniile semantice corespunzătoare.

Interpretări (Evaluări)

```
prog :: Prog -> DomProg
stmt :: Instr -> DomInstr
expr ::  Expr -> DomExpr
```

Observație. Interpretările trebuie să reflecte cerințele semantice explicate în limbaj natural. De exemplu

```
prog :: Prog -> DomProg
prog (On s) = stmt s 0
```

deoarece am precizat că valoarea inițială a celulei de memorie Mem este 0.

Semantica denotațională

Etape în definirea semanticii denotaționale

- identificăm categoriile sintactice;
- asociem fiecărei categorii sintactice un domeniu semantic;
- definim interpretări ale categoriilor sintactice în domeniile semantice.

```
type DomProg = [Int]
type DomInstr = Int -> [Int]
type DomExpr = Int -> Int
prog :: Prog -> DomProg
stmt :: Instr -> DomInstr
expr :: Expr -> DomExpr
```

Semantica denotațională în Haskell

```
type DomProg = [Int]
type DomInstr =  Int -> [Int]
type DomExpr = Int -> Int

prog :: Prog -> DomProg
prog (On s) = stmt s 0

stmt :: Instr -> DomInstr
stmt  (e :> s) m = let v = expr e m in  (v : (stmt s v))
stmt Off _ = []

expr ::  Expr -> DomExpr
expr (e1 :+  e2) m = (expr  e1 m) + (expr  e2 m)
expr (V n) _  =  n
expr  Mem m  = m
```

Mini-Haskell

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcțional și semantica lui denotațională.

- Limbajul Mini-Haskell conține:
  - expresii de tip **Bool** și expresii de tip **Int**
  - expresii de tip funcție (λ-expresii)
  - expresii provenite din aplicarea functiilor
- Pentru a defini semantica limbajului vom introduce domeniile semantice (valorile) asociate expresiilor limbajului.
- Pentru a evalua (interpreta) expresiile vom defini un mediu de evaluare (memoria) in care vom retine variabilele si valorile curente asociate.

Sintaxă

```
type Name = String

data    Hask    =    HTrue
                |    HFalse
                |    HLit Int
                |    Hif Hask Hask Hask
                |    Hask :==: Hask
                |    Hask :+: Hask
                |    HVar Name
                |    HLam Name Hask
                |    Hask :$: Hask
    deriving (Read, Show)
infix  4 :==:
infixl 6 :+:
infixl 9 :$:
```

Domenii semantice

Domeniul valorilor

```
data    Value  =    VBool Bool
               |    VInt Int
               |    VFun (Value -> Value)
               |    VError -- pentru reprezentarea erorilor
```

Mediul de evaluare

```
type    HEnv   =    [(Name, Value)]
```

Domeniul de evaluare

Fiecărei expresii i se va asocia ca denotație o funcție de la medii de evaluare la valori. Deci domeniul de evaluare al expresiilor este

```
type    DomHask    =    HEnv -> Value
```

Afișarea expresiilor din Hask

```
instance Show Value where
  show (VBool b)    = show b
  show (VInt i)     = show i
  show (VFun _)     = "<function>"
  show VError       = "<error>"
```

Observație

Funcțiile nu pot fi afișate efectiv, ci doar generic.

Egalitate pentru valori

```
instance Eq Value where
  (VBool b) == (VBool c)   = b == c
  (VInt i)  == (VInt j)    = i == j
  (VFun _)  == (VFun _)    = error "Unknown"
  VError    == VError      = error "Unknown"
  _         == _           = False
```

Observație

Funcțiile si erorile nu pot fi testate dacă sunt egale.

```
Prelude> :t error
error :: [Char] -> a
-- permite afisarea unui mesaj de eroare pentru orice tip
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
type DomHask = HEnv -> Value
hEval :: Hask -> DomHask

hEval HTrue r          = VBool True
hEval HFalse r         = VBool False

hEval (HLit i) r       = VInt i

hEval (HIf c d e) r    =
  hif (hEval c r) (hEval d r) (hEval e r)
    where
      hif (VBool b) v w = if b then v else w
      hif _ _ _ = VError
```

Evaluarea expresiilor Mini-Haskell în Haskell

hEval :: Hask -> DomHask --*type DomHask = HEnv -> Value*

```
hEval (d :==: e) r =  heq (hEval d r) (hEval e r)
  where
    heq (VInt i) (VInt j) = VBool (i == j)
    heq _ _               = VError
```

```
hEval (d :+: e) r = hadd (hEval d r) (hEval e r)
  where
    hadd (VInt i) (VInt j) = VInt (i + j)
    hadd _ _               = VError
```

Evaluarea variabilelor Mini-Haskell în Haskell

hEval :: Hask -> DomHask --*type DomHask = HEnv -> Value*

- Pentru a evalua o variabilă HVar x trebuie să găsim valoarea asociată lui x în HEnv = [(Name, Value)]
- Putem folosi următoarele funcții predefinite:

```
-- lookup din modulul Data.List
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup _key []          =  Nothing
lookup  key ((x,y):xys)
    | key == x          =  Just y
    | otherwise         =  lookup key xys

-- fromMaybe din modulul Data.Maybe
fromMaybe :: a -> Maybe a -> a
fromMaybe d x = case x of
    Nothing -> d
    Just v  -> v
```

Evaluarea variabilelor Mini-Haskell în Haskell

hEval :: Hask -> DomHask --*type DomHask = HEnv -> Value*

- Pentru a evalua o variabilă HVar x trebuie să găsim valoarea asociată lui x în HEnv = [(Name, Value)]

```
hEval (HVar x) r = fromMaybe VError (lookup x r)
```

Evaluarea λ-expresiilor Mini-Haskell în Haskell

hEval :: Hask -> DomHask --*type DomHask = HEnv -> Value*

```
hEval :: Hask -> HEnv -> Value
```

- abstractizarea (întoarce o valoare de tip VFun)

```
hEval (HLam x e) r      = VFun (\ v -> hEval e ((x,v):r))
```

- aplicarea (aplică o valore de tip VFun)

```
hEval (d :$: e) r       = happ (hEval d r) (hEval e r)
  where
    happ (VFun f) v = f v
    happ _ _        = VError
```

Evaluarea expresiilor Mini-Haskell în Haskell

hEval :: Hask -> DomHask --*type DomHask = HEnv -> Value*

```
hEval HTrue r          = VBool True
hEval HFalse r         = VBool False
hEval (HIf c d e) r    =
    hif (hEval c r) (hEval d r) (hEval e r)
  where hif (VBool b) v w = if b then v else w
        hif _ _ _         = VError
hEval (HLit i) r       = VInt i
hEval (d :=: e) r      = heq (hEval d r) (hEval e r)
  where heq (VInt i) (VInt j) = VBool (i == j)
        heq _ _               = VError
hEval (d :+: e) r    = hadd (hEval d r) (hEval e r)
  where hadd (VInt i) (VInt j) = VInt (i + j)
        hadd _ _               = VError
hEval (HVar x) r       = fromMaybe VError (lookup x r)
hEval (HLam x e) r     = VFun (\ v -> hEval e ((x,v):r))
hEval (d :$: e) r    = happ (hEval d r) (hEval e r)
  where happ (VFun f) v = f v
        happ _ _        = VError
```

Evaluarea expresiilor Mini-Haskell în Haskell

Test

```
h0 =
    (HLam "x" (HLam "y" (HVar "x" :+: HVar "y")))
    :$: HLit 3
    :$: HLit 4

test_h0 = hEval h0 [] == VInt 7
```

```
*Main> test_h0
True
```

Pe săptămâna viitoare!