Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ioana Leuștean
Traian Florin Șerbănuță

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
traian.serbanuta@unibuc.ro

1. Testare - tipuri de date cu valori puține

2. Generarea numerelor pseudo-aleatoare

3. QuickCheck
   - Testable
   - Alți generatori

Testare - tipuri de date cu valori puține

Testare QuickCheck - Exemplu

K. Claessen, J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". Proceedings of the ICFP, ACM SIGPLAN, 2000.

```
import Test.QuickCheck

myreverse :: [Int] -> [Int]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

prdef :: [Int] -> Bool
prdef xs = (myreverse xs == reverse xs)

*Main> quickCheck prdef
+++ OK, passed 100 tests.
```

Testare QuickCheck - Exemplu

```haskell
import Test.QuickCheck

myreverse :: [Int] -> [Int]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

wrongpr :: [Int] -> Bool
wrongpr xs = (myreverse xs == xs)

*Main> quickCheck wrongpr
*** Failed! Falsified (after 4 tests and 3 shrinks):
[1,0]
```

Testare QuickCheck - Exemplu

```
import Test.QuickCheck


myreverseW :: [Int] -> [Int]
myreverseW [] = []
myreverseW (x:xs) = x:(myreverse1 xs)

prdefW :: [Int] -> Bool
prdefW xs = (myreverseW xs == reverse xs)

*Main> quickCheck prW
*** Failed! Falsified (after 4 tests and 5 shrinks):
[0,1]
```

Clasa tipurilor "mici"

http://www.cse.chalmers.se/edu/year/2018/course/TDA452/lectures/OverloadingAndTypeClasses.html

- Definiți clasa tipurilor de date cu un număr "mic" de valori.

```
class MySmall a where
  smallValues :: [a]
```

Clasa tipurilor "mici"

http://www.cse.chalmers.se/edu/year/2018/course/TDA452/lectures/OverloadingAndTypeClasses.html

- Definiți clasa tipurilor de date cu un număr "mic" de valori.

```
class MySmall a where
  smallValues :: [a]

instance MySmall Bool where
   smallValues = [True, False]
```

Clasa tipurilor "mici"

http://www.cse.chalmers.se/edu/year/2018/course/TDA452/lectures/OverloadingAndTypeClasses.html

- Definiți clasa tipurilor de date cu un număr "mic" de valori.

```
class MySmall a where
  smallValues :: [a]

instance MySmall Bool where
   smallValues = [True, False]

data Season = Spring | Summer | Autumn | Winter
              deriving Show
instance MySmall Season  where
   smallValues = [Spring, Summer, Autumn, Winter]

> smallValues :: [Season] -- trebuie sa precizam tipul
[Spring,Summer,Autumn,Winter]
```

Clasa tipurilor "mici"

```
class MySmall a  where
  smallValues :: [a]
```

Clasa tipurilor "mici"

```
class MySmall a  where
  smallValues :: [a]


instance MySmall Int where
   smallValues = [0,12,3,45,91,100]
```

Clasa tipurilor "mici"

```
class MySmall a  where
  smallValues :: [a]


instance MySmall Int where
   smallValues = [0,12,3,45,91,100]

instance (MySmall a, MySmall b) => MySmall (a, b) where
   smallValues = [(v1, v2) | v1 <- smallValues, v2 <-
       smallValues]

> smallValues :: [(Season, Bool)]
[(Spring,True),(Spring,False),(Summer,True),(Summer,False),(
   Autumn,True),(Autumn,False),(Winter,True),(Winter,False)
   ]
```

Clasa tipurilor "mici" - testare

- Definiți o clasa care conține o funcție asemănătoare cu quickCheck care testează dacă o proprietate este adevărată pentru toate valorile unui tip "mic".

```
class  MySmallCheck a where
   smallValues :: [a]
   smallCheck :: (a -> Bool) -> Bool
```

Clasa tipurilor "mici" - testare

- Definiți o clasa care conține o funcție asemănătoare cu quickCheck care testează dacă o proprietate este adevărată pentru toate valorile unui tip "mic".

```
class  MySmallCheck a where
   smallValues :: [a]
   smallCheck :: (a -> Bool) -> Bool
   smallCheck prop = and [ prop x | x <- smallValues ]
   -- minimal definition: smallValues
```

Clasa tipurilor "mici" - testare

```
class  MySmallCheck a where
  smallValues :: [a]
  smallCheck :: (a -> Bool) -> Bool
```

Clasa tipurilor "mici" - testare

```
class  MySmallCheck a where
   smallValues :: [a]
   smallCheck :: (a -> Bool) -> Bool
   smallCheck prop = and [ prop x | x <- smallValues ]

instance MySmallCheck Int where
   smallValues = [0,12,3,45,91,100]


propInt :: Int -> Bool
propInt x = x < 90
propInt1 :: Int -> Bool
propInt1 x = x < 101

> smallCheck propInt
False
> smallCheck propInt1
True
```

Clasa tipurilor "mici" - testare

- Putem defini smallCheck astfel încât să precizeze un contraexemplu?

Clasa tipurilor "mici" - testare

- Putem defini smallCheck astfel încât să precizeze un contraexemplu?

```
class Show a => MySmallCheck a where
  smallValues :: [a]
  smallCheck :: (a -> Bool) -> Bool
  smallCheck prop = sc smallValues
    where
      sc [] = True
      sc (x:xs)
        | prop x     = sc xs
        | otherwise = error ("Counterexample:" ++ show x)
instance MySmallCheck Int where
   smallValues = [0,12,3,45,91,100]

propInt :: Int -> Bool
propInt x = x < 90

> smallCheck propInt
*** Exception: Counterexample:91
```

Generarea numerelor pseudo-aleatoare

PRNG

Ce facem cand avem tipuri cu un numar mare de valori (asa cum este **Int**)? Trebuie să generăm valori pseudo-aleatoare.

PRNG

Un *Pseudo random number generator* este un algoritm care produce o secvența de numere aleatoare, având ca punct de plecare o valoare inițială (*seed*).

Exemplu:

$$\begin{aligned} \text{Linear Congruence Generator:} \quad X_{i+1} &= aX_i + c \ (mod\ m) \\ seed &= X_0 \end{aligned}$$

Exemplu: numere aleatoare între 0 și 10

- Generator de numere aleatoare

```
rval i = (7 * i + 3) `mod` 11  -- valori intre 0 si 10
```

```
> rval 0 -- samanta este 0
3        -- valorea aleatoare generata
```

Exemplu: numere aleatoare între 0 și 10

- Generator de numere aleatoare

```
rval i = (7 * i + 3 ) `mod` 11  -- valori intre 0 si 10

> rval 0 -- samanta este 0
3         -- valorea aleatoare generata
```

- Generăm o secvență de numere aleatoare

```
genRandSeq 0 _ = []
genRandSeq n s = let news = rval s
                 in news : (genRandSeq (n-1) news)

-- n este numarul de valori care vor fi generate
-- s este samanta
```

Exemplu: numere aleatoare între 0 și 10

- Generăm o secvență de numere aleatoare

```
rval i = (7 * i + 3 ) `mod` 11  -- valori intre 0 si 10

genRandSeq 0 _ = []
genRandSeq n s = let news = rval seed
                 in news : (genRandSeq (n-1) news)
```

```
> genRandSeq 10 0
[3,2,6,1,10,7,8,4,9,0]
```

Exemplu: numere aleatoare între 0 și 10

- Generăm o secvență de numere aleatoare

```
rval i = (7 * i + 3 ) `mod` 11  -- valori intre 0 si 10

genRandSeq 0 _ = []
genRandSeq n s = let news = rval seed
                 in news : (genRandSeq (n-1) news)



> genRandSeq 10 0
[3,2,6,1,10,7,8,4,9,0]

> genRandSeq 20 0
[3,2,6,1,10,7,8,4,9,0,3,2,6,1,10,7,8,4,9,0]
```

Exemplu: numere aleatoare între 0 și 10

- Generăm o secvență de numere aleatoare

```
rval i = (7 * i + 3 ) `mod` 11  -- valori intre 0 si 10

genRandSeq 0 _ = []
genRandSeq n s = let news = rval seed
                 in news : (genRandSeq (n-1) news)
```

```
> genRandSeq 10 0
[3,2,6,1,10,7,8,4,9,0]

> genRandSeq 20 0
[3,2,6,1,10,7,8,4,9,0,3,2,6,1,10,7,8,4,9,0]
```

Secvența aleatoare este predictibilă. Cum îmbunătățim algoritmul?

Exemplu: numere aleatoare între 0 și 10

- Folosim generatoare dfierite pentru valori si semințe

```
rval i = (7 * i + 3 ) `mod` 11   -- valori intre 0 si 10
rseed i = (7 * i + 3 ) `mod` 101

genRandSeq 0 _ = []
genRandSeq n s = let
                      val  = rval s
                      news = rseed s
                  in  (val : (genRandSeq (n-1) news) )

> genRandSeq 10 0
[3,2,6,9,10,0,0,8,8,3]
> genRandSeq 20 0
[3,2,6,9,10,0,0,8,8,3,7,2,3,9,5,4,6,6,3,10]
> genRandSeq 30 0
[3,2,6,9,10,0,0,8,8,3,7,2,3,9,5,4,6,6,3,10,9,4,3,6,1,3,4,5,
9,2]
```

PRNG: valorile și semințele sunt diferite

```
type Seed = Int
type RValue = Int

myrand :: Seed -> (RValue, Seed)
myrand i = (rval i, rseed i)

genRandSeq 0 _ = []
genRandSeq n  s = let (val,news) = myrand s
                  in  (val : (genRandSeq (n-1) news) )
```

Generarea numerelor aleatoare în Haskell

http://hackage.haskell.org/package/random-1.1/docs/System-Random.html

```
class RandomGen g where
   next :: g -> (Int,g)
 -- observati asemanarea cu myrand :: Seed ->(RValue, Seed)
   ...

data StdGen
instance RandomGen StdGen where ...

mkStdGen :: Int -> StdGen

---  pt tipuri oarecare
class Random a where
   random :: RandomGen g => g -> (a, g)
   randoms :: RandomGen g => g -> [a]
   randomRs :: RandomGen g => (a, a) -> g -> [a]
   ....
```

Generarea numerelor aleatoare în Haskell

http://hackage.haskell.org/package/random-1.1/docs/System-Random.html

```
System.Random> genInt = fst $ random (mkStdGen 1000) :: Int

System.Random> genInt
1611434616111168504
System.Random> genInt
1611434616111168504

System.Random> genInts = randoms (mkStdGen 500) :: [Int]

System.Random> take 10 genInts
[-8476283234809671955,5851875716463766781,-11743329760464713711

-6005536961401157228,1127019136727650924,-5427348788055872176,

-3587680396420832273,-1231390686875326875,4168674226095003295,

-6936465015900757066]
```

Generarea caracterelor aleatoare în Haskell

http://hackage.haskell.org/package/random-1.1/docs/System-Random.html

```
System.Random> genChar = fst$randomR ('a','z') (mkStdGen
     500)::Char
System.Random> genChar
'x'
System.Random> genChar
'x'
System.Random> genChars = randomRs ('a','z') (mkStdGen 500)::
       [Char]
System.Random> take 10 genChars
"xofmefswxj"
System.Random> take 50 genChars
"xofmefswxjxyhuuuditkpdrrqrhbdsfyyyhtfutowrxlnszfct"
```

QuickCheck

Testable

```
Prelude> import Test.QuickCheck
Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
```

- Pentru a putea fi testată o proprietate trebuie să aparțină unei instanțe a clasei Testable
- Instanță trivială

```
instance Testable Bool where
   ...
```

```
Prelude Test.QuickCheck> quickCheck (1 + 2 == 3)
+++ OK, passed 1 test.
Prelude Test.QuickCheck> quickCheck (1 + 2 == 8)
*** Failed! Falsified (after 1 test):
```

Testable

Instanță interesantă

```
quickCheck :: Testable prop => prop -> IO ()
```

```
instance (Arbitrary a, Show a, Testable prop) =>
  Testable (a -> prop)  where ..
```

- putem defini proprietăți care depind de parametri
- Aproape toate tipurile standard de date sunt instanțe ale lui Arbitrary

Testable

Instanță interesantă

```
quickCheck :: Testable prop => prop -> IO ()

instance (Arbitrary a, Show a, Testable prop) =>
  Testable (a -> prop)  where ..
```

- putem defini proprietăți care depind de parametri
- Aproape toate tipurile standard de date sunt instanțe ale lui Arbitrary

```
Prelude> quickCheck (\x -> x + 0 == x)
+++ OK, passed 100 tests.
Prelude> quickCheck (\x y z -> (x + y) + z == x + (y + z))
+++ OK, passed 100 tests.
Prelude> quickCheck (\x y z -> (x - y) - z == x - (y - z))
*** Failed! Falsified (after 3 tests and 2 shrinks):
0
0
1
```

Testare QuickCheck - Exemplu

```
import Test.QuickCheck

myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]


prdef xs = (myreverse xs == reverse xs)
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.
```

Testare QuickCheck - Exemplu

```
import Test.QuickCheck

myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]


prdef xs = (myreverse xs == reverse xs)
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.

> quickCheck wrongpr
+++ OK, passed 100 tests.
```

Ce se întâmplă?

Testare QuickCheck - Exemplu

```
import Test.QuickCheck
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

> verboseCheck wrongpr
...
Passed:
[(),(),(),(),(),(),(),(),()]
Passed:
[(),(),(),()]
Passed:
[(),(),(),()]
Passed:
[(),(),(),(),(),(),(),(),(),(),(),()]
Passed:
[(),(),(),(),(),(),()]
...
```

Testare QuickCheck - Exemplu

Trebuie să precizăm tipul datelor testate!

```
import Test.QuickCheck
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

prdef :: [Int] -> Bool  -- precizam tipul
prdef xs = (myreverse xs == reverse xs)
wrongpr :: [Int] -> Bool  -- precizam tipul
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.
```

Testare QuickCheck - Exemplu

Trebuie să precizăm tipul datelor testate!

```
import Test.QuickCheck
myreverse :: [a] -> [a] -- definita generic
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++[x]

prdef :: [Int] -> Bool  -- precizam tipul
prdef xs = (myreverse xs == reverse xs)
wrongpr :: [Int] -> Bool  -- precizam tipul
wrongpr xs = myreverse xs == xs

> quickCheck prdef
+++ OK, passed 100 tests.

> quickCheck wrongpr
*** Failed! Falsified (after 4 tests and 3 shrinks):
[1,0]
```

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
                  deriving (Show, Eq)

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)

wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs


> quickCheck prdef1
```

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)

wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs


> quickCheck prdef1
error: No instance for (Arbitrary Season)
```

Să ne aducem aminte că:

```
instance (Arbitrary a, Show a, Testable prop) =>
  Testable (a -> prop) where ..
```

Testare QuickCheck

- Generarea testelor aleatoare depinde de tipul de date.
- Tipurile de date pot fi folosite ca argumente în teste QuickCheck dacă sunt instanțe ale clasei Arbitrary:

```
class Arbitrary a where
  arbitrary :: Gen a
```

- Gen a este un "wrapper" pentru un alt generator

```
newtype Gen a = MkGen{ unGen :: QCGen -> Int -> a}
```

http://hackage.haskell.org/package/QuickCheck-2.13.2/docs/src/Test.QuickCheck.Random.html

http://hackage.haskell.org/package/QuickCheck-2.13.2/docs/Test-QuickCheck.html

Testare QuickCheck

- Tipurile de date pot fi folosite ca argumente în teste QuickCheck dacă sunt instanțe ale clasei Arbitrary:

```
class Arbitrary a where
  arbitrary :: Gen a
```

- Gen a poate fi tratat ca un tip abstract, datele de tip Gen a pot fi definite cu ajutorul combinatorilor:

```
choose :: Random a => (a, a) -> Gen a
oneof :: [Gen a] -> Gen a
elements :: [a] -> Gen a
....
```

https://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck.html#g:8

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

instance Arbitrary Season where
    arbitrary = elements [Spring, Summer, Autumn, Winter]
```

Testare QuickCheck - ADT

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Show, Eq)

instance Arbitrary Season where
    arbitrary = elements [Spring, Summer, Autumn, Winter]

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)
wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs

> quickCheck prdef1
+++ OK, passed 100 tests.

> quickCheck wrongpr1
*** Failed! Falsified (after 3 tests):
[Winter,Summer]
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                deriving (Show, Eq)

instance Arbitrary MyInt where
   arbitrary =  elements (map My listInt)
                where listInt = take 500000 (randoms (
                   mkStdGen 0)) :: [Int]
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                deriving (Show, Eq)

instance Arbitrary MyInt where
   arbitrary =  elements (map My listInt)
                where listInt = take 500000 (randoms (
                   mkStdGen 0)) :: [Int]

prdef1 :: [Season] -> Bool
prdef1 xs = (myreverse xs == reverse xs)
wrongpr1 :: [Season] -> Bool
wrongpr1 xs = myreverse xs == xs

> quickCheck prdef1
+++ OK, passed 100 tests.
> quickCheck wrongpr1
*** Failed! Falsified (after 3 tests):
[Winter,Summer]
```

Testare QuickCheck

instance Testable Property where ...

Property ne poate ajuta să extindem limbajul logic cu combinatori precum:

- (===) :: (**Eq** a, **Show** a) **=>**a –> a –> Property

```
> quickCheck (\x y-> x === y * (x `div` y) + x `mod` y)
*** Failed! Exception: 'divide by zero' (after 1 test):
0
0
Exception thrown while showing test case: 'divide by
    zero'
```

- (==>) :: Testable prop **=>** **Bool** –> prop –> Property

```
> quickCheck (\x y-> y /= 0 ==> x === y * (x `div` y) +
    x `mod` y)
+++ OK, passed 100 tests; 12 discarded.
```

- Ca și **Bool**, Property e o instanță finală, argumentele se adaugă peste
- Mai sunt și alți combinatori, precum forall
  https://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck.html#g:17

Testare QuickCheck

instance (Testable a) => Testable (Maybe a) where ...

- Ca și **==>** de la Property ne poate ajuta să filtrăm cazurile nedefinite

```
testDivMod :: Integer -> Integer -> Maybe Bool
testDivMod _ 0 = Nothing
testDivMod x y = Just $ x == y * (x `div` y) + x `mod` y

Prelude Test.QuickCheck> quickCheck testDivMod
+++ OK, passed 100 tests; 11 discarded.
```

- Poate fi folosit cu orice Testable

```
testDivMod :: Integer -> Maybe (Integer -> Property)
testDivMod 0 = Nothing
testDivMod y =
  Just $ \x -> x === y * (x `div` y) + x `mod` y

Prelude Test.QuickCheck> quickCheck testDivMod
+++ OK, passed 100 tests; 15 discarded.
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                deriving (Show, Eq)
```

Cum definim instanța lui Arbitrary? O variantă ar fi tot folosirea operației elements:

```
instance Arbitrary MyInt where
   arbitrary =  elements (map My listInt)
                where listInt = [-1000000, 1000000]
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
             deriving (Show, Eq)
```

Cum definim instanța lui Arbitrary? O variantă ar fi tot folosirea operației elements:

```
instance Arbitrary MyInt where
  arbitrary = elements (map My listInt)
             where listInt = [-1000000, 1000000]
```

Putem genera lista de întregi:

```
import System.Random
randoms :: RandomGen g => g -> [a]
       -- instance RandomGen StdGen

instance Arbitrary MyInt where
  arbitrary = elements (map My listInt)
           where
          listInt = take 500000(randoms(mkStdGen 0))::[Int]
```

Testare QuickCheck - ADT

```
import System.Random
newtype MyInt = My Int
                     deriving (Show, Eq)

instance Arbitrary MyInt where
    arbitrary =  elements (map My listInt)
                where
              listInt = take 500000(randoms(mkStdGen 0))::[Int]

wrongpr2 :: [MyInt] -> Bool
wrongpr2 xs = myreverse xs == xs

> quickCheck wrongpr2
*** Failed! Falsified (after 5 tests and 1 shrink):
[My 4948157297514287243,My (-2390719447972180436)]
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                deriving (Show, Eq)
```

Putem defini instanța lui `Arbitrary` folosind direct definiția datelor de tip *Gen a*

```
newtype Gen a = MkGen{ unGen :: QCGen -> Int -> a}
```

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                deriving (Show, Eq)
```

Putem defini instanța lui Arbitrary folosind direct definiția datelor de tip *Gen a*

```
newtype Gen a = MkGen{ unGen :: QCGen -> Int -> a}
```

Știind că **Int** este instanță a lui Arbitrary, să definim o instanță pentru MyInt.

Testare QuickCheck - ADT

```
newtype MyInt = My Int
                deriving (Show, Eq)
```

Putem defini instanța lui Arbitrary folosind direct definiția datelor de tip *Gen a*

```
newtype Gen a = MkGen{ unGen :: QCGen -> Int -> a}
```

Știind că **Int** este instanță a lui Arbitrary, să definim o instanță pentru MyInt.

```
instance Arbitrary MyInt where
  arbitrary =  MkGen (\s i -> let x = f s i in (My x))
                where
                  f = (unGen  (arbitrary :: Gen Int))
```

Pe săptămâna viitoare!