

# Tulajdonságalapú tesztelés

# QuickCheck

A QuickCheck Haskell programok automatikus, tulajdonság-alapú tesztelésére használható.

- ▶ *Programspecifikáció*: program által teljesítendő tulajdonságok
- ▶ Nagy számú, a *tulajdonságok alapján generált* tesztesetek
- ▶ A specifikáció *Haskellben adható meg*, kombinátorokkal
- ▶ További *kombinátorok*: tulajdonságok definiálása, eredmények elemzése, tesztadatok előállítása
- ▶ Megvalósítás: beágyazott szakterületspecifikus nyelv (*embedded domain-specific language*), amely viszont tetszőleges nyelven írt program teszteléséhez használható!

*"The coolest example of type classes that I know"* – Simon Peyton Jones ("Adventure with Types in Haskell",

<https://www.youtube.com/watch?v=6COvD8oynmI>)

# A QuickCheck (*Test.QuickCheck*) használata

```
flickSort :: Ord α => [α] → [α]
flickSort [] = []
flickSort (x : y : xs) = (y : x : xs)
flickSort xs = sort xs
-- prop_idempotent :: [Int] → Bool
prop_idempotent xs = flickSort (flickSort xs) ≡ flickSort xs
```

Tesztelés:

```
-- quickCheck :: Testable prop => prop -> IO ()
GHCi> quickCheck prop_idempotent
+++ OK, passed 100 tests.
```

További tulajdonságok:

```
prop_ordered xs = ordered (flickSort xs)
  where ordered xs = and (zipWith (<) xs (tail xs))

prop_sortModel xs = (sort xs) ≡ (flickSort xs)

prop_append xs ys = not (null xs) ⊃ not (null ys) ⊃
  head (flickSort (xs ++ ys)) ≡ (minimum xs) 'min' (minimum ys)
```

Bővebben: <http://hackage.haskell.org/package/QuickCheck>

# Tesztelés tetszőleges adattípussal

```
data Ternary = Yes | No | Unknown  
  deriving (Show, Eq)
```

```
instance Arbitrary Ternary where  
  -- arbitrary = elements [Yes, No, Unknown]  
  arbitrary = do  
    n ← choose (0, 2) :: Gen Int  
    return $ case n of  
      0 → Yes  
      1 → No  
      _ → Unknown
```

## Tesztelés tetszőleges adattípussal (folytatás)

```
data Tree  $\alpha$  = Node  $\alpha$  (Tree  $\alpha$ ) (Tree  $\alpha$ ) | Leaf  
  deriving (Show, Eq)
```

```
instance Arbitrary  $\alpha \Rightarrow$  Arbitrary (Tree  $\alpha$ ) where  
  arbitrary = sized tree where
```

```
    tree 0 = return Leaf
```

```
    tree n = do
```

```
      elem  $\leftarrow$  arbitrary
```

```
      lt     $\leftarrow$  tree (n `div` 2)
```

```
      rt     $\leftarrow$  tree (n `div` 2)
```

```
      return (Node elem lt rt)
```

```
shrink Leaf = []
```

```
shrink (Node x l r) =
```

```
  [Leaf] ++
```

```
  [l, r] ++
```

```
  [Node x' l' r' | (x', (l', r'))  $\leftarrow$  shrink (x, (l, r))]
```

# Nem üres listák generálása (példa)

```
newtype NonEmptyList  $\alpha$  = NEL [ $\alpha$ ]  
  deriving (Eq, Ord, Show)
```

```
instance Arbitrary  $\alpha \Rightarrow$  Arbitrary (NonEmptyList  $\alpha$ ) where  
  arbitrary = do  
    xs  $\leftarrow$  arbitrary `suchThat` (not  $\circ$  null)  
    return (NEL xs)
```

Kipróbálás:

```
GHCi> sample (arbitrary :: Gen (NonEmptyList Int))
```

Alternatív megoldás:

```
instance Arbitrary  $\alpha \Rightarrow$  Arbitrary (NonEmptyList  $\alpha$ ) where  
  arbitrary = sized list where  
    list n = do  
      xs  $\leftarrow$  forM [0..n] ( $\lambda$  _ . arbitrary)  
      return (NEL xs)
```

# Implementáció: A *Gen* monád

```
-- StdGen, Random, randomR, split: System.Random  
newtype Gen  $\alpha$  = G (StdGen  $\rightarrow$  Int  $\rightarrow$   $\alpha$ )
```

```
runGen :: Gen  $\alpha$   $\rightarrow$  StdGen  $\rightarrow$  Int  $\rightarrow$   $\alpha$   
runGen (G f) r n = f r n
```

```
instance Functor Gen where  
    fmap f x = G $ \ r n . f (runGen x r n)
```

```
instance Applicative Gen where  
    pure x    = G $ \ _ _ . x  
    f <*> x = G $ \ r n . (runGen f r n) (runGen x r n)
```

```
instance Monad Gen where  
    x >>= f = G $ \ r n .  
        let (r1, r2) = split r in  
        let x'         = f (runGen x r1 n) in  
        runGen x' r2 n
```

```
choose :: Random  $\alpha$   $\Rightarrow$  ( $\alpha$ ,  $\alpha$ )  $\rightarrow$  Gen  $\alpha$   
choose rng = G $ \ r _ . fst (randomR rng r)
```

# Implementáció: A *Gen* monád (folytatás)

*oneof* :: [*Gen*  $\alpha$ ]  $\rightarrow$  *Gen*  $\alpha$

*oneof* [] = *error* "Empty list"

*oneof* *gs* = **do**

*i*  $\leftarrow$  *choose* (0, (*length* *gs*) - 1)

*gs* !! *i*

*elements* :: [ $\alpha$ ]  $\rightarrow$  *Gen*  $\alpha$

*elements* *xs* = *oneof* (*map* *return* *xs*)

*sized* :: (*Int*  $\rightarrow$  *Gen*  $\alpha$ )  $\rightarrow$  *Gen*  $\alpha$

*sized* *f* = *G* \$  $\lambda$  *r n* . *r* *runGen* (*f* *n*) *r n*

*sample'* :: *Gen*  $\alpha$   $\rightarrow$  *IO* [ $\alpha$ ]

*sample'* *m* = **do**

*rnd*<sub>0</sub>  $\leftarrow$  *newStdGen*

**let** *rnds* *rnd* = *rnd*<sub>1</sub> : *rnds* *rnd*<sub>2</sub> **where** (*rnd*<sub>1</sub>, *rnd*<sub>2</sub>) = *split* *rnd*

*return* [ *runGen* *m* *r* *n* | (*r*, *n*)  $\leftarrow$  (*rnds* *rnd*<sub>0</sub>) `zip` [0, 2 ... 20] ]

*sample* :: (*Show*  $\alpha$ )  $\Rightarrow$  *Gen*  $\alpha$   $\rightarrow$  *IO* ()

*sample* *g* = **do**

*samples*  $\leftarrow$  *sample'* *g*

*forM* \_ *samples* *print*



# Implementáció: Az *Arbitrary* osztály

**class** *Arbitrary*  $\alpha$  **where**

*arbitrary* :: *Gen*  $\alpha$

*arbitrary* = *error* "No default generator"

*shrink* ::  $\alpha \rightarrow [\alpha]$

*shrink* \_ = []

**instance** *Arbitrary* () **where**

*arbitrary* = *return* ()

**instance** *Arbitrary Bool* **where**

*arbitrary* = *choose* (*False*, *True*)

*shrink* *True* = [*False*]

*shrink* *False* = []

# Implementáció: Az *Arbitrary* osztály (egészek)

**instance Arbitrary Integer where**

*arbitrary* = sized \$ \lambda n . **do**

**let**  $n'$  = *toInteger*  $n$

*choose* ( $-n'$ ,  $n'$ )

*shrink*  $x$  = *nub* ( $[-x \mid x < 0, -x > x]$  ++ *others*)

**where**

*others* = *takeWhile* ( $<< x$ ) *approx*

*approx* = ( $0 : [x - i \mid i \leftarrow \text{tail } (\text{iterate } (\text{quot } 2) x)]$ )

$a << b$  = **case** ( $a \geq 0, b \geq 0$ ) **of**

  (*True*, *True*)     $\rightarrow a < b$

  (*False*, *False*)  $\rightarrow a > b$

  (*True*, *False*)    $\rightarrow a + b < 0$

  (*False*, *True*)    $\rightarrow a + b > 0$

# Implementáció: Az *Arbitrary* osztály (párok és listák)

**instance** (*Arbitrary*  $\alpha$ , *Arbitrary*  $\beta$ )  $\Rightarrow$  *Arbitrary*  $(\alpha, \beta)$  **where**

*arbitrary* = **do**

$x \leftarrow \text{arbitrary}$

$y \leftarrow \text{arbitrary}$

**return**  $(x, y)$

*shrink*  $(x, y) = [(x', y) \mid x' \leftarrow \text{shrink } x]$

$\text{++ } [(x, y') \mid y' \leftarrow \text{shrink } y]$

**instance** *Arbitrary*  $\alpha \Rightarrow$  *Arbitrary*  $[\alpha]$  **where**

*arbitrary* = **sized** \$  $\lambda n$  . **do**

$k \leftarrow \text{choose } (0, n)$

**forM**  $[1..k]$   $(\lambda \_ . \text{arbitrary})$

*shrink*  $[] = []$

*shrink*  $(x : xs) = [xs]$

$\text{++ } [x : xs' \mid xs' \leftarrow \text{shrink } xs]$

$\text{++ } [x' : xs \mid x' \leftarrow \text{shrink } x]$

# Függvények generálása

A *Gen* bővítése:

```
promote :: Monad  $\mu \Rightarrow \mu (Gen \alpha) \rightarrow Gen (\mu \alpha)$   
promote m = G $  $\lambda r n . do$   
  g  $\leftarrow$  m  
  return (runGen g r n)  
  
variant :: Integer  $\rightarrow Gen \alpha \rightarrow Gen \alpha$   
variant k0 g = G ( $\lambda r n . runGen g (var k_0 r) n$ ) where  
  var k r | k  $\equiv$  k' = r'  
          | otherwise = var k' r'  
  where  
    (r1, r2) = split r  
    r' | even k = r1  
        | otherwise = r2  
    k' = k `div` 2
```

Az *Arbitrary* bővítése:

```
class CoArbitrary  $\alpha$  where  
  coarbitrary ::  $\alpha \rightarrow Gen \gamma \rightarrow Gen \gamma$   
  
(><) :: ( $Gen \alpha \rightarrow Gen \alpha$ )  $\rightarrow (Gen \alpha \rightarrow Gen \alpha) \rightarrow (Gen \alpha \rightarrow Gen \alpha)$   
(><) f g gen = do  
  n  $\leftarrow$  arbitrary  
  (g  $\circ$  variant n  $\circ$  f) gen
```

# Függvények generálása (folytatás)

```
instance (CoArbitrary  $\alpha$ , Arbitrary  $\beta$ )  $\Rightarrow$  Arbitrary ( $\alpha \rightarrow \beta$ ) where  
  arbitrary = promote (\coarbitrary arbitrary)
```

```
instance CoArbitrary Bool where  
  coarbitrary False = variant 0  
  coarbitrary True  = variant (-1)
```

```
instance CoArbitrary Integer where  
  coarbitrary = variant
```

```
instance (CoArbitrary  $\alpha$ , CoArbitrary  $\beta$ )  $\Rightarrow$  CoArbitrary ( $\alpha, \beta$ ) where  
  coarbitrary (x, y) = coarbitrary x >< coarbitrary y
```

```
instance CoArbitrary  $\alpha \Rightarrow$  CoArbitrary [ $\alpha$ ] where  
  coarbitrary []      = variant 0  
  coarbitrary (x : xs) = variant (-1)  $\circ$  coarbitrary (x, xs)
```

Például:

```
GHCi> fs <- sample' (arbitrary :: Gen (Integer -> Integer))  
GHCi> map (\f -> f 0) fs  
[0,0,-4,-1,7,10,10,5,6,-6,-1]
```

# Implementació: A *Property* tipus

```
type Result    = Maybe Bool -- Nothing: discarded  
type LGen      = WriterT String Gen  
type Property  = LGen Result  
runProperty :: Property → Gen (Result, String)  
runProperty = runWriterT
```

```
class Testable  $\pi$  where  
  property ::  $\pi \rightarrow$  Property
```

```
instance Testable () where  
  property _ = return Nothing
```

```
instance Testable Bool where  
  property b = return (Just b)
```

```
instance Testable Result where  
  property = return
```

# Implementáció: A *Property* típus (folytatás)

**instance** (*Testable*  $\pi$ )  $\Rightarrow$  *Testable* (*LGen*  $\pi$ ) **where**

*property*  $p$  = **do**

$x \leftarrow p$

*property*  $x$

**instance** (*Arbitrary*  $\alpha$ , *Show*  $\alpha$ , *Testable*  $\pi$ )  $\Rightarrow$  *Testable* ( $\alpha \rightarrow \pi$ ) **where**

*property*  $f$  = **do**

$x \leftarrow \text{lift arbitrary}$

*tell* (*show*  $x$ )

*property* ( $f\ x$ )

**infixr** 0  $\supset$

$(\supset) :: (\text{Testable } \pi) \Rightarrow \text{Bool} \rightarrow \pi \rightarrow \text{Property}$

*False*  $\supset \_ = \text{property } ()$

*True*  $\supset p = \text{property } p$

*toGen*  $:: (\text{Testable } \pi) \Rightarrow \pi \rightarrow \text{Gen}(\text{Result}, \text{String})$

*toGen* = *runProperty*  $\circ$  *property*

# Implementáció: A meghajtó (driver)

```
check :: (Testable  $\pi$ )  $\Rightarrow$   $\pi \rightarrow IO ()$ 
check p = do
  rnd  $\leftarrow$  newStdGen
  let (maxRun, maxDiscarded) = (100, 100)
  (n, d)  $\leftarrow$  test (maxRun, maxDiscarded, 0, rnd) (runGen (toGen p))
  let (run, discarded) = (100 - n, 100 - d)
  putStrLn (show run ++ " tests run, " ++ show discarded ++ " discarded.")

test :: (Int, Int, Int, StdGen)
       $\rightarrow$  (StdGen  $\rightarrow$  Int  $\rightarrow$  (Result, String))  $\rightarrow IO (Int, Int)$ 
test (0, d, _, _) _ = return (0, d)
test (n, 0, _, _) _ = return (n, 0)
test (n, d, s, seed) f = do
  let (rnd1, rnd2) = split seed
  let (p, t)         = f rnd1 s
  case p of
    Nothing     $\rightarrow$  test (n, d - 1, s + 1, rnd2) f
    Just True   $\rightarrow$  test (n - 1, d, s + 1, rnd2) f
    Just False  $\rightarrow$  do
      putStrLn ("Failed : " ++ t)
      return (n, d)
```