

Automatické generování testů dle specifikace (QuickCheck)

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2018

Testování dle konkrétních hodnot

- testujeme jednotlivé součásti samostatně (*unit testing*)
- porovnáváme výsledky na modelových datech
- modelová data i výsledky na nich si musíme vytvořit ručně

Testování dle konkrétních hodnot

- testujeme jednotlivé součásti samostatně (*unit testing*)
 - porovnáváme výsledky na modelových datech
 - modelová data i výsledky na nich si musíme vytvořit ručně
-
- + testují přesně ty případy, které chceme
 - + jednoduché na přípravu
 - časově náročné na přípravu
 - pokrývají jen ty možnosti, na které si vzpomeneme
 - testují jenom konkrétní případy, ne všeobecné chování

Testování dle konkrétních hodnot

Například balík HUnit:

```
import Test.HUnit
```

```
runTests :: IO Counts
```

```
runTests = runTestTT $ TestList [testSet1, testSet2]
```

```
testSet1 :: Test
```

```
testSet1 = TestLabel "Factorials" $
```

```
  TestList [ fact 4 ~?= 25, fact 0 ~?= 1 ]
```

```
testSet2 :: Test
```

```
testSet2 = TestLabel "Numerical functions" $
```

```
  TestList [ even 4 ~? "4 even?", odd 4 ~? "4 odd?" ]
```

Testování dle specifikace

- Chtěli bychom testovat specifikaci, ne konkrétní případy!
- Chtěli bychom, aby se testy generovaly automaticky!
- Chtěli bychom pěkné (pokud možno minimální) protipříklady!

Testování dle specifikace

- Chtěli bychom testovat specifikaci, ne konkrétní případy!
 - Chtěli bychom, aby se testy generovaly automaticky!
 - Chtěli bychom pěkné (pokud možno minimální) protipříklady!
- ⇒ Dodejme specifikaci pomocí invariantů.
- Dělejme testy na platnost invariantů v konkrétních případech.
- ⇒ Případy generujme náhodně.
- Nejsou některé hodnoty zajímavější pro testy než jiné?
 - Jak vybírat náhodně v nekonečných doménách?
- ⇒ Po nalezení protipříkladu se ho pokusme zmenšit.

Za invariant považujeme nějakou vlastnost (*property*), která je univerzálně platná. V Haskellu ji můžeme zapsat třeba jako predikát.

Co je invariantem v následujících situacích?

- `max :: Int -> Int -> Int`

Za invariant považujeme nějakou vlastnost (*property*), která je univerzálně platná. V Haskellu ji můžeme zapsat třeba jako predikát.

Co je invariantem v následujících situacích?

- `max :: Int -> Int -> Int`
- `take :: Int -> [a] -> [a]`

Hledání invariantů

Za invariant považujeme nějakou vlastnost (*property*), která je univerzálně platná. V Haskellu ji můžeme zapsat třeba jako predikát.

Co je invariantem v následujících situacích?

- `max :: Int -> Int -> Int`
- `take :: Int -> [a] -> [a]`
- `insertSort :: Ord a => [a] -> [a]`
`insert :: Ord a => a -> [a] -> [a]`

Hledání invariantů

Za invariant považujeme nějakou vlastnost (*property*), která je univerzálně platná. V Haskellu ji můžeme zapsat třeba jako predikát.

Co je invariantem v následujících situacích?

- `max :: Int -> Int -> Int`
- `take :: Int -> [a] -> [a]`
- `insertSort :: Ord a => [a] -> [a]`
`insert :: Ord a => a -> [a] -> [a]`
- funkce z domácího úkolu 1

The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases.

- balík QuickCheck
(<https://hackage.haskell.org/package/QuickCheck>)
- moduly `Test.QuickCheck.*`
- typicky stačí importovat `Test.QuickCheck`

Funkce pro samotné testování:

- `quickCheck :: Testable prop => prop -> IO ()`
- `verboseCheck :: Testable prop => prop -> IO ()`
- `quickCheckWith :: Testable prop =>`
`Args -> prop -> IO ()`
`stdArgs :: Args`

Příklad

```
prop_basic1 :: Int -> [a] -> Bool  
prop_basic1 n xs = length (take n xs) == n
```

Příklad

```
prop_basic1 :: Int -> [a] -> Bool  
prop_basic1 n xs = length (take n xs) == n
```

```
prop_basic2 :: Int -> [a] -> Bool  
prop_basic2 n xs = length (take n xs) <= n
```

Příklad

```
prop_basic1 :: Int -> [a] -> Bool
prop_basic1 n xs = length (take n xs) == n
```

```
prop_basic2 :: Int -> [a] -> Bool
prop_basic2 n xs = length (take n xs) <= n
```

```
prop_basic3 :: NonNegative Int -> [a] -> Bool
prop_basic3 (NonNegative n) xs =
  length (take n xs) <= n
```

Generátory náhodných prvků

```
newtype Gen a = MkGen { unGen :: QCGen -> Int -> a }  
sample :: Show a => Gen a -> IO ()
```

- Gen a představuje generátor náhodných hodnot typu a
- generátor je vlastně funkce náhodného generátoru (v konkrétním stavu) a parametru velikosti, která vrátí prvek požadovaného typu
- existuje standardní instance Monad Gen

Typová třída Arbitrary

```
class Arbitrary a where  
  arbitrary :: Gen a  
  shrink   :: a -> [a]
```

- typy, pro které je možno vygenerovat „náhodný prvek“
- arbitrary je náhodný generátor pro daný typ
- shrink se používá při zmenšování protipříkladů

Typová třída Arbitrary

```
class Arbitrary a where  
  arbitrary :: Gen a  
  shrink   :: a -> [a]
```

- typy, pro které je možno vygenerovat „náhodný prvek“
- arbitrary je náhodný generátor pro daný typ
- shrink se používá při zmenšování protipříkladů

```
class CoArbitrary a where  
  coarbitrary :: a -> Gen b -> Gen b
```

- typy, které mohou být argumenty „náhodných funkcí“

Příklad

```
data Pack = EmptyPack      -- empty pack
          | Tomatoes Double -- tomato weight in kg
          | Cucumbers Int   -- number of cucumbers
deriving (Eq, Show)
```

Příklad

```
data Pack = EmptyPack          -- empty pack
          | Tomatoes Double    -- tomato weight in kg
          | Cucumbers Int      -- number of cucumbers
          deriving (Eq, Show)
```

```
packGen1 :: Gen Pack
packGen1 = oneof [ return EmptyPack
                  , fmap Tomatoes arbitrary
                  , fmap Cucumbers arbitrary ]
```

```
instance Arbitrary Pack where
  arbitrary = packGen1
```

Příklad

```
checkout :: [Pack] -> Double
checkout = sum . map price
  where price EmptyPack = 0
        price (Tomatoes weight) = weight * 33.50
        price (Cucumbers count) =
          fromIntegral count * 19.90

prop_pack1 :: [Pack] -> Bool
prop_pack1 pack = checkout pack >= 0
```

Příklad

```
prop_pack2 :: [Pack] -> Property
prop_pack2 pack =
  all nonNegative pack ==> checkout pack >= 0
  where nonNegative (Tomatoes w) = w >= 0
        nonNegative (Cucumbers n) = n >= 0
        nonNegative _ = True
```

Příklad

```
prop_pack2 :: [Pack] -> Property
prop_pack2 pack =
  all nonNegative pack ==> checkout pack >= 0
  where nonNegative (Tomatoes w) = w >= 0
        nonNegative (Cucumbers n) = n >= 0
        nonNegative _ = True

packGen2 :: Gen Pack
packGen2 = oneof
  [ return EmptyPack
  , fmap Tomatoes (arbitrary `suchThat` (>=0))
  , fmap Cucumbers (arbitrary `suchThat` (>=0)) ]
```

Příklad

```
data BinTree = BEmpty | BNode Int BinTree BinTree
              deriving (Eq, Ord, Show)
```

```
instance Arbitrary BinTree where
  arbitrary = treeGen1
  shrink    = treeShrink
```

```
treeGen1 :: Gen BinTree
treeGen1 = oneof [ return BEmpty,
                  liftM3 BNode arbitrary treeGen1 treeGen1 ]
```

```
treeShrink :: BinTree -> [BinTree]
treeShrink BEmpty = []
treeShrink (BNode v l r) = [l, r]
                          ++ [ BNode v' l r | v' <- shrink v ]
                          ++ [ BNode v l' r | l' <- shrink l ]
                          ++ [ BNode v l r' | r' <- shrink r ]
```



```
treeToList :: BinTree -> [Int]
treeToList BEmpty = []
treeToList (BNode v l r) = v : treeToList l ++
    treeToList r

treeSum :: BinTree -> Int
treeSum BEmpty = 0
treeSum (BNode v l r) = v + treeSum l + treeSum r

prop_tree1 :: BinTree -> Bool
prop_tree1 t = treeSum t == sum (treeToList t)
```

```
prop_tree2 :: BinTree -> Property
prop_tree2 t = classify (treeSize t == 0) "trivial"
  $ prop_tree1 t
```

```
prop_tree2 :: BinTree -> Property
prop_tree2 t = classify (treeSize t == 0) "trivial"
  $ prop_tree1 t
```

```
prop_tree3 :: BinTree -> Property
prop_tree3 t = classify (treeSize t == 1) "easy" $
  prop_tree2 t
```

```
prop_tree2 :: BinTree -> Property
prop_tree2 t = classify (treeSize t == 0) "trivial"
  $ prop_tree1 t
```

```
prop_tree3 :: BinTree -> Property
prop_tree3 t = classify (treeSize t == 1) "easy" $
  prop_tree2 t
```

```
prop_tree4 :: BinTree -> Property
prop_tree4 t = collect (treeSize t) $ prop_tree3 t
```

Příklad

```
treeSize :: BinTree -> Int
treeSize BEmpty = 0
treeSize (BNode _ l r) = 1 + treeSize l + treeSize r
```

```
treeSize :: BinTree -> Int
treeSize BEmpty = 0
treeSize (BNode _ l r) = 1 + treeSize l + treeSize r

treeGen3 :: Gen BinTree
treeGen3 = sized treeGen where
  treeGen 0 = return BEmpty
  treeGen n = frequency
    [ (1, return BEmpty)
    , (4, liftM3 BNode arbitrary subtree subtree) ]
    where subtree = treeGen (n `div` 2)
```

QuickCheck – užitečné funkce

- `(=>)` :: Testable prop =>
 Bool -> prop -> Property
- `(===)` :: (Eq a, Show a) => a -> a -> Property
- `forall` :: (Show a, Testable prop) =>
 Gen a -> (a -> prop) -> Property
- `classify` :: Testable prop =>
 Bool -> String -> prop -> Property
- `collect` :: (Show a, Testable prop) =>
 a -> prop -> Property

Tvorba nových generátorů:

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

Vybrané funkce pracující s generátory:

- `listOf :: Gen a -> Gen [a]`
- `vectorOf :: Int -> Gen a -> Gen [a]`
- `oneof :: [Gen a] -> Gen a`
- `frequency :: [(Int, Gen a)] -> Gen a`
- `sized :: (Int -> Gen a) -> Gen a`
- `suchThat :: Gen a -> (a -> Bool) -> Gen a`

Možnost nechat QuickCheck najít všechny testy:

```
{-# LANGUAGE TemplateHaskell #-}  
-- all the module code  
-- at the end:  
return []
```

```
runTests :: IO Bool  
runTests = $quickCheckAll
```

- první řádek zapíná rozšíření *TemplateHaskell* (musí být před hlavičkou modulu)
- `$quickCheckAll` je funkce, která za *kompilace* vloží kód pro spuštění všech testů pojmenovaných `prop_*`
- `return []` způsobí, že `$quickCheckAll` může najít testy (které jsou před tímto řádkem)

Testování funkcí

```
prop_fold :: Fun (Int, Int) Int -> Int -> [Int]
           -> Property
prop_fold f z xs = foldr (applyFun2 f) z xs
                  === foldl (applyFun2 f) z x
```

Testování funkcí

```
prop_fold :: Fun (Int, Int) Int -> Int -> [Int]
           -> Property
prop_fold f z xs = foldr (applyFun2 f) z xs
                 === foldl (applyFun2 f) z x
```

```
> quickCheck prop_fold
*** Failed! Falsifiable (after 3 tests and 16
    shrinks):
{(0,2)->0, _->1}
2
[0]
0 /= 1
```

- Fun a b je generovatelná funkce z a do b
- je možné ji zobrazit (po případech/vzorech), provádět shrink
- více viz Test.QuickCheck.Function

Rekapitulace – QuickCheck

- + Testujeme program vůči specifikaci, ne konkrétním případům.
- + Testy můžou najít i případy, které by nás nenapadly.
- + Konkrétní případy pro testy jsou generovány automaticky.
- + Donutí nás důkladněji se zamyslet nad specifikací.

Rekapitulace – QuickCheck

- + Testujeme program vůči specifikaci, ne konkrétním případům.
 - + Testy můžou najít i případy, které by nás nenapadly.
 - + Konkrétní případy pro testy jsou generovány automaticky.
 - + Donutí nás důkladněji se zamyslet nad specifikací.
-
- Všechno stojí a padá na dobré volbě invariantů.
 - Potřebujeme vhodný generátor náhodných prvků.
 - Přesná specifikace je často příliš složitá.
 - Jestliže nspecifikujeme invarianty přesně a implementujeme špatný generátor, můžeme dostat falešný pocit bezpečí!
 - Nemusí vždy nalézt neobvyklé chyby, chyby na okrajových případech – může dávat různé odpovědi!

QuickCheck je silný nástroj, musíme však:

- rozumět, co dělá,
- a především rozumět, co nedělá!

QuickCheck je silný nástroj, musíme však:

- rozumět, co dělá,
- a především rozumět, co nedělá!

... a pak můžeme vybudovat třeba `hsExprTest`

QuickCheck je silný nástroj, musíme však:

- rozumět, co dělá,
- a především rozumět, co nedělá!

... a pak můžeme vybudovat třeba `hsExprTest`

- obdoba QuickCheck existuje i v mnoha jiných (i imperativních) programovacích jazycích

- 1 Pro datový typ `Filesystem` nadefinovaný v souboru `Task04a.hs` napište náhodný generátor. Následně s pomocí knihovny `QuickCheck` otestujte, že funkce `numFiles1` a `numFiles2` se chovají stejně.
- 2 S pomocí knihovny `QuickCheck` otestujte vybrané invarianty v první domácí úloze.