Testování dle specifikace, QuickCheck

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2016

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

(Za ukázku testů děkujeme Ondrovi Mosnáčkovi.)

```
main = hspec $ do
  describe "addVertex" $ do
    it "addVertex (Vertex 1) testGraph" $ addVertex
    (Vertex 1) testGraph `shouldBe` testGraph
  describe "findVertex" $ do
    it "findVertex (Vertex 5) testGraph" $
    findVertex (Vertex 5) testGraph `shouldBe`
    M.fromList
       [(Edge {from = Vertex 4, to = Vertex 5},6),
        (Edge {from = Vertex 5, to = Vertex 4},6),
        (Edge {from = Vertex 5, to = Vertex 6},9),
        (Edge {from = Vertex 6, to = Vertex 5},9)]
```

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!

$\Rightarrow$ Dodejme specifikaci pomocí invariantů.
  - Dělejme testy na platnost invariantů v konkrétních případech.

$\Rightarrow$ Případy generujme náhodně.
  - Nejsou některé hodnoty zajímavější pro testy než jiné?
  - Jak vybírat náhodně v nekonečných doménách?

$\Rightarrow$ Po nalezení protipříkladu se ho pokusme zmenšit.

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`

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]`

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]`
- moduly z domácích úkolů 1 a 2

QuickCheck

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`

QuickCheck - užitečné funkce

Funkce pro samotné testování:

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

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 vrací prvek požadovaného typu
- existuje standardní instance `Monad Gen`

Typová třída Arbitrary

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

- zahrnuje typy, z kterých je možné vygenerovat „náhodný prvek“
- `arbitrary` je náhodný generátor pro daný typ
- `shrink` je funkce, která se používá při zmenšování protipříkladů
- existuje i třída `CoArbitrary`, která slouží pro generování náhodných funkcí

Příklad

```
newtype NonNegativeInt = NonNegativeInt Int
                         deriving Show

nonNegativeGen = fmap NonNegativeInt $
                   choose (0, 50 :: Int)

instance Arbitrary NonNegativeInt where
  arbitrary = nonNegativeGen
```

Příklad

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

instance Arbitrary Pack where
  arbitrary = packGen1

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

Příklad

```
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

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

Příklad

```
treeGen2 :: Gen BinTree
treeGen2 = frequency
  [ (1, return BEmpty)
  , (4, liftM3 BNode arbitrary treeGen2 treeGen2) ]
```

Příklad

```
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`

Generátory náhodných prvků - užitečné funkce

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`

Rekapitulace - QuickCheck

+ Testujeme program vůči specifikaci, ne vůči 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 nespecifikujeme 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!

Úkol na cvičení

1. Pro datový typ `Filesystem` nadefinovaný v souboru `Task07a.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 druhé domácí úloze (haldy).