Testování v Haskellu: HUnit, QuickCheck

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

HUnit: Jednotkové testování

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
\+ stačí základní znalost jazyka
− 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 ~?= 24, fact 0 ~?= 1 ]

testSet2 :: Test
testSet2 = TestLabel "Numerical functions" $
  TestList [ even 4 ~? "4 even?", odd 4 ~? "4 odd?" ]
```

QuickCheck: Testování dle specifikace

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.

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]`
- `filter :: (a -> Bool) -> [a] -> [a]`

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`

Příklad

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

Příklad

```haskell
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

```haskell
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

- **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**

```
sample :: Show a => Gen a -> IO ()
```

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`

Typová třída Arbitrary

```haskell
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
- výsledný seznam **nesmí** obsahovat zadaný prvek

```haskell
-- default shrink implementation
shrink _ = []
```

Příklad

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

Příklad

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

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

instance Arbitrary Pack where
  arbitrary = packGen1
```

Příklad

```haskell
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

```haskell
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

```haskell
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
  [ pure EmptyPack
  , fmap Tomatoes (arbitrary `suchThat` (>=0))
  , fmap Cucumbers (arbitrary `suchThat` (>=0)) ]
```

Příklad

```haskell
data BinTree = Empty | Node Int BinTree BinTree
             deriving (Eq, Ord, Show)

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

Příklad

```haskell
data BinTree = Empty | Node Int BinTree BinTree
             deriving (Eq, Ord, Show)

instance Arbitrary BinTree where
  arbitrary = treeGen1
  shrink = treeShrink

treeGen1 :: Gen BinTree
```

Příklad

```haskell
data BinTree = Empty | Node Int BinTree BinTree
             deriving (Eq, Ord, Show)

instance Arbitrary BinTree where
  arbitrary = treeGen1
  shrink = treeShrink

treeGen1 :: Gen BinTree
treeGen1 = oneof [ pure Empty, liftM3 Node arbitrary treeGen1 treeGen1 ]
```

Příklad

```haskell
data BinTree = Empty | Node Int BinTree BinTree
             deriving (Eq, Ord, Show)

instance Arbitrary BinTree where
  arbitrary = treeGen1
  shrink = treeShrink

treeGen1 :: Gen BinTree
treeGen1 = oneof [ pure Empty, liftM3 Node arbitrary treeGen1 treeGen1 ]


treeShrink :: BinTree -> [BinTree]
```

Příklad

```haskell
data BinTree = Empty | Node Int BinTree BinTree
             deriving (Eq, Ord, Show)

instance Arbitrary BinTree where
  arbitrary = treeGen1
  shrink = treeShrink

treeGen1 :: Gen BinTree
treeGen1 = oneof [ pure Empty, liftM3 Node arbitrary treeGen1 treeGen1 ]


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

Příklad

```haskell
-- Create inorder list of values.
treeToList :: BinTree -> [Int]

-- Calculate sum of all nodes in tree.
treeSum :: BinTree -> Int

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

Příklad

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

Příklad

```haskell
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
```

Příklad

```haskell
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

```haskell
-- Calculate size of tree.
treeSize :: BinTree -> Int

treeGen3 :: Gen BinTree
treeGen3 = sized treeGen where
  treeGen 0 = pure Empty
  treeGen n = frequency
    [ (1, pure Empty)
    , (4, liftM3 Node 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`

QuickCheck – užitečné modifikátory

- `NonZero a`
  `(Num a, Eq a, Arbitrary a) => Arbitrary (NonZero a)`
- `Positive a`
  `(Num a, Ord a, Arbitrary a) => Arbitrary (Positive a)`
- `NonEmptyList a`
  `Arbitrary a => Arbitrary (NonEmptyList a)`
- `InfiniteList a`
  `Arbitrary a => Arbitrary (InfiniteList a)`
- `SortedList a`
  `(Arbitrary a, Ord a) => Arbitrary (SortedList a)`
- `Small a`
  `Integral a => Arbitrary (Small a)`

A další, viz modul `Test.QuickCheck.Modifiers`.

Hromadné testování

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

```haskell
-- Property to test if 'foldr' and 'foldl' calculates the same things.
prop_fold :: (Int -> Int -> Int) -> Int -> [Int] -> Property
prop_fold f z xs = foldr f z xs === foldl f z xs
```

Testování funkcí

```haskell
-- Property to test if 'foldr' and 'foldl' calculates the same things.
prop_fold :: (Int -> Int -> Int) -> Int -> [Int] -> Property
prop_fold f z xs = foldr f z xs === foldl f z xs
```

Nefunguje, protože funkce nejsou v typové třídě **Show**. Lze opravit pomocí modulu **Text.Show.Functions**, ale výsledek není moc uspokojivý.

Testování funkcí

```haskell
-- Property to test if 'foldr' and 'foldl' calculates the same things.
prop_fold :: (Int -> Int -> Int) -> Int -> [Int] -> Property
prop_fold f z xs = foldr f z xs === foldl f z xs
```

Nefunguje, protože funkce nejsou v typové třídě **Show**. Lze opravit pomocí modulu **Text.Show.Functions**, ale výsledek není moc uspokojivý.

Řešení: **QuickCheck** definuje datový typ **Fun** a b pro funkce a -> b, které jde generovat, vypisovat a zmenšovat. Pracuje se s nimi pomocí:

- `applyFun :: Fun a b -> a -> b`
- `applyFun2 :: Fun (a, b) c -> a -> b -> c`
- ...

Testování funkcí

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

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 xs
```

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

Generování funkcí – implementace

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

- typy, které můžou být argumenty „náhodných funkcí“
- pro detaily viz např.
  https://begriffs.com/posts/2017-01-14-design-use-quickcheck.html

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 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!

Rekapitulace – QuickCheck

QuickCheck je silný nástroj, musíme však:

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

Rekapitulace – QuickCheck

QuickCheck je silný nástroj, musíme však:

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

. . . a pak můžeme vybudovat třeba hsExprTest

Rekapitulace – QuickCheck

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

Samostatné programování

HUnit: merge

Chceme naprogramovat funkci `merge :: Ord a => [a] -> [a] -> [a]`, která pro dvě vstupní seřazené posloupnosti vrátí seřazenou posloupnost, která obsahuje právě prvky dvou vstupních posloupností. Tedy máme kostru

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs ys = undefined
```

Pomocí nástroje **HUnit** napište vhodné jednotkové testy.

Poté funkci `merge` implementujte a ověřte, jestli vaše testy prochází.

QuickSort: merge

Napište vhodné invarianty pro předchozí funkci `merge` a pomocí nástroje **QuickCheck** ověřte, jestli je vaše implementace splňuje.

Možná by se vám mohl hodit modifikátor **Test.QuickCheck.SortedList** a a funkce

```
isSorted :: Ord a => [a] -> Bool
isSorted xs = and $ zipWith (<=) xs (tail xs)
```

Quick(Check|Sort)

Mějme následující implementaci algoritmu QuickSort:

```
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (pivot:rest) = quickSort (filter (<pivot) rest) ++
                         [pivot] ++
                         quickSort (filter (>pivot) rest)
```

Napište vhodné invarianty a pomocí nástroje **QuickCheck** ověřte, jestli je tato implementace korektní.

Pokud korektní není, algoritmus opravte, aby všechny vaše invarianty splňoval.

QuickCheck: rekurzivní datová struktura

Napište instanci typové třídy **Arbitrary** pro následující datový typ.

```
data Filesystem = Folder [Filesystem] | File deriving Show
```

Pomocí této instance ověřte, jestli jsou následující dvě funkce ekvivalentní:

```
numFiles1 :: Filesystem -> Int
numFiles1 File = 1
numFiles1 (Folder xs) = sum $ map numFiles1 xs

numFiles2 :: Filesystem -> Int
numFiles2 f = let (folders, files) = cnt f in files
 where
   cnt File = (0, 1)
   cnt (Folder f) = foldr (\(x,y) (a,b) -> (x+a,y+b)) (1,0) (map cnt f)
```

QuickCheck: komplikovanější instance Arbitrary

Mějme následující datový typ reprezentující binární vyhledávací stromy:

```
data BST = Leaf Int | Node Int BST BST deriving Show
```

Napište pro datový typ **BST** instanci typové třídy **Arbitrary**, která generuje jen korektní binární vyhledávací stromy.