QuickCheck

Malte Heins und Stefan Wahlen

Einführung: QuickCheck

Überprüfung der Korrektheit von Programmen

Formaler Beweis bei größeren Programmen in der Regel zu aufwendig

Automatisierte Massentests leistungsfähiger als manuelle Einzeltests

Erzeugung von zufälligen Testdaten anhand formaler Spezifikationen

Eigenschaften in QuickCheck

Einfache Eigenschaft:

```
prop_PlusAssociative x y z = (x + y) + z == x + (y + z)
```

Testen der Eigenschaft:

```
quickCheck prop_PlusAssociative
```

Wichtig: Typsignatur!

```
prop_PlusAssociative :: Integer -> Integer -> Integer -> Bool
```

Eigenschaften in QuickCheck

Sortiertes Einfügen in Listen

```
prop_InsertOrdered x xs = ordered (insert x xs)
```

Warum schlägt dies fehl?

Eigenschaften in QuickCheck

Sortiertes Einfügen in Listen

```
prop_InsertOrdered x xs = ordered (insert x xs)
```

Warum schlägt dies fehl?

- Test funktioniert nur, wenn auch die Liste, in die eingefügt werden soll, sortiert ist

Eigenschaften in QuickCheck

Test nur mit sortierten Listen

Vorbedingung:

```
prop_InsertOrdered x xs = ordered xs ==> ordered (insert x xs)
```

Generator:

```
prop_InsertOrdered x = forAll orderedList $

                                    \xs -> ordered (insert x xs)
```

Testen mit Abstrakten Datentypen

```
type Queue a = [a]
empty        = []
add x q      = q ++ [x]
isEmpty q    = null q
front (x:q)  = x
remove (x:q) = q
```

Testen mit Abstrakten Datentypen

```
type Queue a = [a]
empty        = []
add x q      = q ++ [x]
isEmpty q    = null q
front (x:q)  = x
remove (x:q) = q
```

```
type Queue2 a     = ([a], [a])
empty2            = ([], [])
add2 (f, b)       = (f, x:b)
isEmpty2 (f, b)   = null f
front2 (x:f, b)   = x
remove2 (x:f, b) = flipQ (f, b)
  where
    flipQ ([], b) = (reverse b, [])
    flipQ q       = q
```

Testen mit Abstrakten Datentypen

Was benötigen wir, um die beiden Implementierungen auf Gleichheit zu prüfen?

Testen mit Abstrakten Datentypen

Was benötigen wir, um die beiden Implementierungen auf Gleichheit zu prüfen?

```
retrieve      :: Queue2 Integer -> Queue Integer
retrieve (f,b) = f ++ reverse b
```

Testen mit Abstrakten Datentypen

Was benötigen wir, um die beiden Implementierungen auf Gleichheit zu prüfen?

```
prop_empty      = retrieve empty2 == empty
prop_add   x q  = retrieve (add2 x q) == add x (retrieve q)
prop_isEmpty q  = isEmpty2 q == isEmpty (retrieve q)
prop_front   q  = front2 q == front (retrieve q)
prop_remove  q  = retrieve (remove2 q) == remove (retrieve q)
```

Testen mit Abstrakten Datentypen

Invariante: Der Beginn der Queue ist nur dann leer, wenn auch das Ende der Queue leer ist

```
invariant (f,b) = not (null f) || null b
```

Testen mit Abstrakten Datentypen

Invariante: Der Beginn der Queue ist nur dann leer, wenn auch das Ende der Queue leer ist

```
invariant (f,b) = not (null f) || null b

prop_empty      = retrieve empty2 == empty
prop_add   x q  = invariant q ==>
                  retrieve (add2 x q) == add x (retrieve q)
prop_isEmpty q  = invariant q ==>
                  isEmpty2 q == isEmpty (retrieve q)
prop_front   q  = invariant q ==> front2 q == front (retrieve q)
prop_remove  q  = invariant q ==>
                  retrieve (remove2 q) == remove (retrieve q)
```

Testen mit Abstrakten Datentypen

front2 und remove2 dürfen nicht mit leeren Listen getestet werden

Testen mit Abstrakten Datentypen

front2 und remove2 dürfen nicht mit leeren Listen getestet werden

```
prop_front q = invariant q && not (isEmpty2 q)
               ==> front2 q == front (retrieve q)

prop_remove q = invariant q && not (isEmpty2 q)
                ==> retrieve (remove2 q) == remove (retrieve q)
```

Testen mit Abstrakten Datentypen

Produzieren die Operationen überhaupt korrekte Queues?

```
prop_inv_empty     = invariant empty2
prop_inv_remove q  = invariant q && not (isEmpty2 q)
                     ==> invariant(remove2 q)
prop_inv_add x q   = invariant q ==> invariant (add2 x q)
```

Testen mit Abstrakten Datentypen

```
add2 (f, b)       = (f, x:b)
```

Was läuft bei add2 falsch?

Testen mit Abstrakten Datentypen

```
add2 (f, b)       = (f, x:b)
```

Was läuft bei add2 falsch?

- add2 fügt am Ende der Liste an und entspricht bei leeren Listen nicht der Invarianten

Testen mit Abstrakten Datentypen

```
alt:  add2 (f, b)       = (f, x:b)
```

Was läuft bei add2 falsch?

- add2 fügt am Ende der Liste an und entspricht bei leeren Listen nicht der Invarianten
  - => `add2 x (f,b) = flipQ(f, x:b)`

Testen mit Abstrakten Datentypen

Algebraische Spezifikationen

```
prop_isEmpty2 q  = invariant q ==> isEmpty2 q == (q == empty2)

prop_front_empty x  = front2 (add2 x empty2) == x

prop_front_add x q  = invariant q && not (isEmpty2 q)
==> front2 (add2 x q) == front2 q

prop_remove_empty x  = remove2 (add2 x empty2) == empty2

prop_remove_add x q  = invariant q && not (isEmpty2 q)
==> remove2 (add2 x q) == add2 x (remove2 q)
```

Testen mit Abstrakten Datentypen

Ist prop_remove_add wirklich falsch?

Testen mit Abstrakten Datentypen

Ist prop_remove_add wirklich falsch?

Nein, die Queues sind äquivalent

Testen mit Abstrakten Datentypen

Ist prop_remove_add wirklich falsch?

Nein, die Queues sind äquivalent

```
q `equiv` q' = invariant q && invariant q' &&
               retrieve q == retrieve q'
```

Testen mit Abstrakten Datentypen

Ist prop_remove_add wirklich falsch?

Nein, die Queues sind äquivalent

```
q `equiv` q' = invariant q && invariant q' &&
               retrieve q == retrieve q'

prop_remove_add x q = invariant q && not (isEmpty2 q)
    ==> remove2 (add2 x q) `equiv` add2 x (remove2 q)
```

Testen mit Abstrakten Datentypen

Als Vorbedingung ist 'equiv' leider nicht pratikabel.

Es werden zwei zufällig Queues erzeugt und stets nur äquivalente benutzt.

Das ist jedoch sehr unwahrscheinlich.

Generatoren

Bisher Vorbedingung:

Viele Testfälle werden erstellt und werden am Ende gar nicht benutzt.

Nun Generatoren:

Es werden explizit korrekte Testfälle generiert.

Generatoren

```
equivQ :: Queue2 Integer -> Gen (Queue2 Integer)
equivQ q = do k <- choose (0, 0 `max` (n-1))
              return (take (n-k) els, reverse(drop (n-k) els))
              where
                els = retrieve q
                n = length els
```

Generatoren

Vorbedingung:

```
prop_add_equiv q q' x  = q `equiv` q'
                       ==> add2 x q `equiv` add2 x q'
```

Generator:

```
prop_add_equiv q q' x   = invariant q
==> forAll (equivQ q) $ \q' -> add2 x q `equiv` add2 x q'
```

Generatoren

Verzweigungen

```haskell
-- ein zufällig bestimmter Generator
oneof :: [Gen a] -> Gen a
oneof  = frequency . map (\x -> (1,x))

-- mit Gewichtung der Wahrscheinlichkeiten
frequency :: [(Int, Gen a)] -> Gen a
```

...und viele mehr

Generatoren für eigene Datentypen

QuickCheck definiert Generatoren für alle eingebauten Datentypen in Haskell

Eigene Datentypen müssen Instanzen der Klasse *Arbitrary* sein, damit QuickCheck Testdaten für diese erzeugen kann

Beispiel: Eigener Datentyp

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

Höhere Gewichtung der Zweige, um triviale Bäume zu vermeiden:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency [
                  (1, return Leaf),
                  (3, liftM3 Branch arbitrary arbitrary arbitrary)
                ]
```

Beispiel: Eigener Datentyp

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

Höhere Gewichtung der Zweige, um triviale Bäume zu vermeiden:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency [
                  (1, return Leaf),
                  (3, liftM3 Branch arbitrary arbitrary arbitrary)
                ]
```

Problem: Viele Bäume terminieren nicht

Beispiel: Eigener Datentyp

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

Beschränkung der Baumgröße:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree

arbTree 0 = return Leaf
arbTree n | n > 0 = frequency[
                        (1, return Leaf),
                        (3, liftM3 Branch shrub arbitrary shrub)
                    ]
    where
        shrub = arbTree (n `div` 2)
```

Teststatistiken

Definition von Eigenschaften zur Bewertung der Testfälle

```
prop_InsertOrdered     :: Integer -> [Integer] -> Bool
prop_InsertOrdered x xs = ordered xs ==>
                          trivial  length xs <= 2) $
                          ordered (insert x xs)


-- vorgegeben:
trivial p = classify p "trivial"
```

Teststatistiken

Definition von Eigenschaften zur Bewertung der Testfälle

```
prop_InsertOrderec      :: Integer -> [Integer] -> Bool
prop_InsertOrdered x xs = ordered xs ==>
                          classify  null xs) "empty lists" $
                          classify  length xs == 1) "unit lists" $
                          ordered (insert x xs)
```

Teststatistiken

Kategorisierung der Testergebnisse

```
prop_InsertOrderec      :: Integer -> [Integer] -> Bool
prop_InsertOrdered x xs = ordered xs ==> collect (length xs) $
                                         ordered (insert x xs)
```

Fazit

QuickCheck als Hilfe zum Verifizieren

Fehlende Vorbedingungen und Invarianten werden erst beim Testen entdeckt

Einfaches Generieren von zufälligen Testfällen

Testen einzelner Funktionalitäten

Noch Fragen?

Noch Fragen?

Danke für Eure Aufmerksamkeit!