Heutiges Thema

Testen von Programmen

- Spezifikationsbasiert
- Werkzeuggestützt
- Automatisch

Fragestellungen

Wie können wir uns (hinreichend) Sicherheit verschaffen, dass...

- unsere Programme *korrekt* sind,
- anderer Leute Programme *korrekt* sind?

Antworten

- Verifikation
  - Formaler Korrektheitsbeweis (Spezifikationskorrektheit, Implementierungskorrektheit)
  - Hohe Sicherheit, hoher Aufwand
- Testen
  - Varianten: Systematisch, ad hoc
  - Kontrollierbarer Aufwand, unbestimmte Qualitätsaussage

    Denke an:
    "Testen kann nur die Existenz von Fehlern zeigen, nicht deren Nichtexistenz" (Dijkstra)

Beobachtung

Testen ist...

- oft überraschend erfolgreich im Aufdecken von Fehlern

Anforderungen

Rechenschaftslegung über...

- Was wurde getestet?
- Wie tiefgehend, wie umfangreich wurde getestet?
- Wie wurde *Erfolg* festgelegt?

Zusätzlich wünschenswert...

- Wiederholbarkeit von Tests
- Wiederholtes Testen nach Programmänderungen

Voraussetzung

Unabdingbar...

- Spezifikation der Programmbedeutung
  - Informell (Kommentar im Programm, in separater Dokumentation)
    $\leadsto$ ...oft mehrdeutig, offen für Auslegungen
  - Formal
    $\leadsto$ ...präzise Semantik, eindeutig

Im folgenden

Spezifikationsbasiertes, werkzeuggestütztes Testen in und für Haskell

- QuickCheck
  - definiert eine *formale Spezifikationssprache*
    $\leadsto$ ...erlaubt Eigenschaftsdefinition direkt im (Haskell-)Quellcode
  - definiert eine *Testfallgenerationssprache*
    $\leadsto$ ...erlaubt eine einfache und kompakte Beschreibung einer großen Zahl von Testfällen
  - erlaubt automatisches Testen aller in einem Modul spezifizierten Eigenschaften einschließlich Fehlerberichts

Bemerkung

Spezifikations- und Testfallgenerationssprache sind...

- Beispiele sog. *domain-specific embedded languages*
  $\leadsto$ ...besondere Stärke funktionaler Programmierung
- implementiert als Kombinatorbibliotheken in Haskell
  $\leadsto$ ...erlaubt die volle Mächtigkeit von Haskell in Eigenschafts- und Testfallgenerationsdefinition
- Bestandteil der Haskellstandarddistribution im Modul `QuickCheck`
  $\leadsto$ ...gewährleistet einfache und direkte Benutzbarkeit

Literaturhinweis

Die folgende Darstellung orientiert sich an...

- Koen Claessen, John Hughes. *Specification-based Testing with QuickCheck*. In Jeremy Gibbons, Oege de Moor (Hrsg.), *The Fun of Programming*. Palgrave MacMillan, 2003.

Weitere im Kontext relevante Literaturstellen...

- Koen Claessen, John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proceedings of the ACM SIGPLAN 2000 International Conference on Functional Programming (ICFP 2000), 268 - 279, 2000.
- Koen Claessen, John Hughes. *Testing Monadic Code with QuickCheck*. In Proceedings ACM SIGPLAN 2002 Haskell Workshop, 2002.

Eigenschaftsdefinition in QuickCheck 1(2)

Erste Beipiele...

Im Programm:

```
prop_PlusAssociative :: Int -> Int -> Int -> Bool
prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
```

In Hugs:

```
Main>quickCheck prop_PlusAssociative
OK, passed 100 tests
```

Bemerkung:

- Typspezifikation für `prop_PlusAssociative` wg. Überladung von `+` erforderlich (sonst Fehlermeldung)
- Typspezifikation ermöglicht typspezifische Testfallgenerierung
- Eigenschaftsdefinition in einfachsten Fällen wie hier: Wahrheitswertfunktion, d.h. Funktion mit Booleschem Resultat

Eigenschaftsdefinition in QuickCheck 2(2)

Im Programm:

```
prop_PlusAssociative :: Float -> Float -> Float -> Bool
prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
```

In Hugs (möglicherweise, denke an Rundungsfehler):

```
Main>quickCheck prop_PlusAssociative
Falsifiable, after 13 tests:
1.0
-5.16667
-3.71429
```

Bemerkung:

Fehlerbericht enthält:

- Anzahl erfolgreich durchgeführter Tests
- Gegenbeispiel

Komplexere Eigenschaftsdefinitionen in QuickCheck 1(3)

Betrachte als zu überprüfende Eigenschaft (eine Funktion `insert` gegeben vorausgesetzt): sortiertes Einfügen in Listen

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

...zu streng/naiv, da `xs` nicht als sortiert vorausgesetzt wird.

Komplexere Eigenschaftsdefinitionen in QuickCheck 2(3)

Abhilfe:

```
prop_InsertOrdered :: Integer -> [Integer] -> Property
prop_InsertOrdered x xs = ordered xs ==> ordered (insert x xs)
```

Bemerkung:

- `ordered xs ==>`: Ergänzen einer *Vorbedingung*

  ⤳ ...Testfälle, die der Vorbedingung nicht genügen, werden ausgeschieden
- `==>`: Kein Boolescher Operator; Operator, der die Auswahl von Testfällen beeinflusst

  ⤳ ...Eigenschaftsdefinitionen, die sich auf solche Operatoren abstützen, haben stets den Typ `Property`

Komplexere Eigenschaftsdefinitionen in QuickCheck 2(3)

Auch möglich:

- Direkte Quantifizierung über geordnete Listen

```
prop_InsertOrdered :: Integer -> Property
prop_InsertOrdered x =
        forAll orderedLists $ \xs -> ordered (insert x xs)
```

Weitergehend:

- Verfeinerung der Spezifikation, so dass (bis auf das eingefügte Element) Übereinstimmung der Resultatliste mit der Ausgangsliste gegeben ist

Ein ausgedehnteres Beispiel

...abstrakter Datentyp für *Schlangen*.

Einfache (aber ineffiziente) Realisierung:

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


Effizientere Realisierung

Zugrundeliegende Idee

- Aufspalten der Liste in Listenanfang und -rest
- Listenrest dabei in umgekehrter Reihenfolge
- Dies gewährleistet: Effizienter Zugriff auf Listenanfang und -ende

```
type QueueI a   = ([a],[a])
emptyI          = ([],[])
addI x (f,b)    = (f,x:b)
isEmptyI (f,b)  = null f
frontI (x:f,b)  = x
removeI (x:f,b) = flipQ (f,b)
  where
    flipQ ([],b) = (reverse b, [])
    flipQ q      = q
```

Zshg. von Listenrepräsentation und dargestellter abstrakter Liste

…vermöge einer *Retrieve*-Funktion:

```
retrieve :: QueueI Integer -> [Integer]
retrieve (f,b) = f ++ reverse b
```

- …überführt die (i.a. zahlreichen) Repräsentationen einer abstrakten Liste als Werte von `QueueI` in die zugrundeliegende abstrakte Liste als Wert von `Queue`
- …die Festlegung von `QueueI` und `Queue` als Listen über ganzen Zahlen erspart uns in der Folge Typspezifikationen für Eigenschaftsdefinitionen

Korrektheitseigenschaften für die Funktionen über QueueI

…mittels `retrieve` überprüfbar, ob

- die Resultate der Anwendung der effizienten Funktionen über `QueueI` mit denen der abstrakten Funktionen über `Queue` übereinstimmen

Korrektheitseigenschaften: Erster Versuch

Folgende Eigenschaften werden demnach als gültig erwartet:

```
prop_empty     = retrieve emptyI == empty
prop_add x q   = retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = isEmptyI q == isEmpty (retrieve q)
prop_front q   = frontI q == front (retrieve q)
prop_remove q  = retrieve (removeI q) == remove (retrieve q)
```


Korrektheitseigenschaften: Erster Versuch

Überprüfung mittels `QuickCheck` liefert:

```
Main>quickCheck prop_isEmpty
Falsifiable, after 4 tests:
([],[-1])
```

Problem:

- Die Spezifikation von `isEmpty` (wie auch von `frontI` und `removeI`) setzt implizit folgende *Invariante* voraus:
  - Der Listenanfang ist höchstens dann leer, wenn auch der Listenrest leer ist

Korrektheitseigenschaften: Verbesserung

Wir definieren folgende Invariante:

```
invariant :: QueueI Integer -> Bool
invariant (f,b) = not (null f) || null b
```

und ergänzen diese in allen Eigenschaftsdefinitionen:

```
prop_empty     = retrieve emptyI == empty
prop_add x q   = invariant q ==>
                   retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = invariant q ==>
                   isEmptyI q == isEmpty (retrieve q)
prop_front q   = invariant q ==>
                   frontI q == front (retrieve q)
prop_remove q  = invariant q ==>
                   retrieve (removeI q) == remove (retrieve q)
```


Korrektheitseigenschaften: Verbesserung

Überprüfung mittels `QuickCheck` liefert:

```
Main>quickCheck prop_isEmpty
OK, passed 100 tests
```

Aber:

```
Main>quickCheck prop_front
Program error: front ([],[])
```

Problem:

- `frontI` (wie auch `removeI`) dürfen nur auf nichtleere Listen angesetzt werden; bislang nicht berücksichtigt.
  - Abhilfe: Ergänze `not (isEmptyI q)` in den Vorbedingungen der entsprechenden Eigenschaften

Korrektheitseigenschaften: Verbesserung

Wir erhalten:

```
prop_empty     = retrieve emptyI == empty
prop_add x q   = invariant q ==>
                   retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q = invariant q ==>
                   isEmptyI q == isEmpty (retrieve q)
prop_front q   = invariant q && not (isEmptyI q) ==>
                   frontI q == front (retrieve q)
prop_remove q  = invariant q && not (isEmptyI q) ==>
                   retrieve (removeI q) == remove (retrieve q)
```

Bemerkung:

- Alle Eigenschaften jetzt mit erfolgreichem Test!

Korrektheitseigenschaften: Noch offen

Bisher getestet:

- Operationen auf Schlangen verhalten sich korrekt auf Repräsentationen von Schlangen, die der Invariante

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

  genügen.

Es bleibt zu überprüfen:

- Schlangenerzeugende Operationen erzeugen Schlangen, die dieser Invariante genügen.

Korrektheitseigenschaften: Noch offen

Die Formulierung entsprechender Eigenschaften für die schlangenproduzierenden Funktionen:

```
prop_inv_empty    = invariant emptyI
prop_inv_add x q  = invariant q ==> invariant (addI x q)
prop_inv_remove q = invariant q && not (isEmptyI q) ==>
                        invariant (removeI q)
```


Korrektheitseigenschaften: Noch offen

Überprüfung mittels QuickCheck liefert:

```
Main>quickCheck prop_inv_add
Falsifiable, after 0 tests:
0
([],[])
```

Problem:

- Invariante muss gelten
  - nicht nur nach Anwendung von removeI,
  - sondern auch nach Anwendung von addI auf die leere Liste

Korrektheitseigenschaften: Noch offen

Dazu:

- Korrektur der Funktion addI wie folgt:

```
addI x (f,b) = flipQ(f,x:b)  -- statt: addI x (f,b) = (f,x:b)
```

Bemerkung:

- Alle Eigenschaften jetzt mit erfolgreichem Test!

Beobachtung

Bei der Entwicklung dieses Beispiels zeigte sich:

- Mittels Testen (nur) ein Fehler in der Implementierung gefunden (bei der Funktion addI)
- *Aber*: Mehrere fehlende Vorbedingungen und eine fehlende Invariante in den ursprünglichen Eigenschaftsdefinitionen gefunden und nachgetragen

Beides ist typisch und wertvoll zugleich:

- Die zusätzlichen Vorbedingungen und Invarianten sind jetzt im Programmtext explizit getroffen
- Sie tragen zu besserem Programmverständnis bei und stellen wertvolle Dokumentation dar, sowohl für den Entwickler wie für spätere Benutzer

Algebraische Spezifikationen

...als (oft gewünschte) Alternative zu abstraktem Modell

Eine algebraische Spezifikation...

- spezifiziert Gleichungen, denen die Operationen genügen sollten

Algebraische Spezifikationen

Für das Beispiel der Schlangen z.B. wie folgt:

```
prop_isEmpty q      = invariant q ==> isEmptyI q == (q == emptyI)
prop_front_empty x  = frontI (addI x emptyI) == x
prop_front_add x q  = invariant q && not (isEmptyI q) ==>
                          frontI (addI x q) == frontI q
prop_remove_empty x = removeI (addI x emptyI) == emptyI
prop_remove_add x q = invariant q && not (isEmptyI q) ==>
                          removeI (addI x q) == addI x (removeI q)
```


Algebraische Spezifikationen

Überprüfung mittels QuickCheck liefert:

```
Main>quickCheck prop_remove_add
Falsifiable, after 1 tests:
0
([1],[0])
```

Problem:

- Linke Seite liefert: ([0,0],[])
- Rechte Seite liefert: ([0],[0])
- Gleichwertig, aber nicht gleich!

Algebraische Spezifikationen

Lösung:

- Übergang von "gleich" zu "äquivalent"

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

und Ersetzen von

```
prop_remove_add x q = invariant q && not (isEmptyI q) ==>
                      removeI (addI x q) == addI x (removeI q)
```

durch

```
prop_remove_add x q = invariant q && not (isEmptyI q) ==>
                      removeI (addI x q) 'equiv' addI x (removeI q)
```

zeitigt das gewünschte Ergebnis.

Algebraische Spezifikationen

Ähnlich wie zuvor bleibt zu überprüfen:

- Alle Schlangenerzeugenden Operationen erzeugen aus äquivalenten Argumenten äquivalente Resultate

Beispielsweise ist für die Operation `addI` zu testen:

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


Algebraische Spezifikationen

Mathematisch korrekt, aber ungeeignet für automatisches Testen, denn:

```
Main>quickCheck prop_add_equiv
Arguments exhausted after 58 tests.
```

Problem und Hintergrund:

- `QuickCheck` erzeugt zufällig Listen `q` und `q'`
- Die meisten der so erzeugten Paare von Listen werden nicht äquivalent sein und für den eigentlichen Test verworfen
- `QuickCheck` erzeugt nur eine bestimmte Zahl von Kandidatenargumenten und bricht dann ab, eventuell bevor 100 Testfälle erreicht sind

Quantifizierungen über Teilmengen

Für `QuickCheck` gilt:

- Parameter sind standardmäßig über alle Werte des entsprechenden Typs quantifiziert

Oft möchte man jedoch:

- Eine Quantifizierung über eine Teilmenge dieser Werte

Quantifizierungen über Teilmengen

Dafür bietet `QuickCheck` mehrere Möglichkeiten:

- Darstellung der Teilmenge mittels *Wahrheitswertfunktionen*, die als Filter für Testfälle dienen
  - Angemessen, wenn sehr viele Elemente der Grundmenge auch in der relevanten Teilmenge sind
  - Unangemessen, wenn nur wenige Elemente der Grundmenge auch in der relevanten Teilmenge sind
- Darstellung der Teilmenge mittels *Generatoren*
  - Ein Generator vom Typ `Gen a` liefert eine zufällige Folge von Werten vom Typ `a`
  - Die Eigenschaft `forall set p` testet `p` sukzessive für zufällig erzeugte Elemente aus `set`

Unterstützung von QuickCheck

Für den effektiven Einsatz von Generatoren unterstützt `QuickCheck`:

- unterschiedlicher Varianten zur Spezifikation von Relationen wie etwa `equiv`
  - Als Wahrheitswertfunktion
    * einfach Äquivalenz zweier Werte zu überprüfen, schwierig äquivalente Werte zu erzeugen
  - Als Funktion von einer Menge in eine Menge äquivalenter Werte (Generator!)
    * einfach äquivalente Werte zu erzeugen, schwierig Äquivalenz zweier Werte zu überprüfen

Generatoren

Generatorvariante für `equiv`:

```
equivQ :: QueueI a -> Gen(QueueI a)
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
```

Bemerkung:

- Definition von `choose` später

Generatoren

Wir können etwa überprüfen:

- Die Gleichwertigkeit der beiden Varianten von `equiv`

Teste dazu:

```
prop_EquivQ q = invariant q ==>
   forAll (equivQ q) $ \q' -> q 'equiv' q'
```

Bemerkung:

- $ entspricht der Funktionsanwendung und erlaubt, auf die Klammerung des λ-Ausdrucks zu verzichten
- Die zu `prop_EquivQ` duale Eigenschaft lässt sich durch Testen nicht zeigen

Generatoren

Damit z.B. möglich:

- Reformulierung der Eigenschaft, dass `addI` äquivalente Schlangen auf äquivalente Schlangen abbildet

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

Bemerkung:

- Andere Eigenschaften analog

Definition von Generatoren

...erleichtert aufgrund des monadischen Typs von `Gen`.

Es gilt:

- `return a` liefert (generiert) stets `a` und repräsentiert die einelementige Menge {`a`}
- `do {x <- s; e}` kann als die (generierte) Menge {`e | x` ∈ `s`} aufgefasst werden

Definition von Generatoren

Die grundlegende Funktion, eine Auswahl zu treffen:

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

Bemerkung:

- Die Funktion `choose` erzeugt "zufällig" ein Element des angegebenen Bereichs
- `choose (1,n)` stellt die Menge {`1...n`} dar

Anwendung von choose

Mithilfe von `choose` läßt sich `equivQ` definieren (wie schon oben gesehen):

```
equivQ :: QueueI a -> Gen(QueueI a)
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
```

- Erzeugt eine Zufallsschlange mit denselben Elementen wie `q`
- Die Anzahl der Elemente im Listenrest wird echt kleiner als die Gesamtzahl der Listenelemente gewählt (sofern diese Anzahl von 0 verschieden ist)

Typbasierte Generatordefinition

...mithilfe des überladenen Generators `arbitrary`

Beispiel:

```
prop_max_le x y = x <= x 'max' y
```

ist äquivalent zu

```
prop_max_le = forAll arbitrary $ \x -> forAll arbitrary $ \y ->
   x <= x 'max' y
```


Typbasierte Generatordefinition

Weiteres Beispiel:

Die Menge {`y | y` ≥ `x` } läßt sich erzeugen durch

```
atLeast x = do diff <- arbitrary
               return (x + abs diff)
```

wegen der (für numerische Typen gültigen) Gleichheit

$$\{y \mid y \geq x\} = \{x + abs\ d \mid d \in \mathbb{Z}\}$$

(Bem.: Ähnliche Festlegung für andere Typen möglich.)

Auswahl

...mithilfe eines Generators `oneof`, der sich als Mengenvereinigung vorstellen lässt.

Beispiel: Konstruktion einer sortierten Liste

```
orderedLists = do x <- arbitrary
                  listsFrom x
   where
     listsFrom x = oneof [return [], do y <- atLeast x
                                        liftM (x:) (listsFrom y)]
```

Intuition dabei:

- Eine sortierte Liste ist entweder leer oder die Hinzufügung eines neuen Anfangselements zu einer sortierten Liste größerer Elemente

Gewichtete Auswahl

- Der `oneof`-Kombinator wählt mit gleicher Wahrscheinlichkeit zwischen den Alternativen
- Oft unerwünschte Auswirkung auf die Testfallgenerierung (zu oft wird die leere Menge gewählt)
- *Abhilfe*: Eine Gewichtsfunktion `frequency`, die den verschiedenen Alternativen Gewichte zuweist

```
frequency :: [(Int,Gen a)] -> Gen a
```

Anwendung:

```
listsFrom x = frequency [(1,return []),
                         (4,do y <- atLeast x
                               liftM (x:) (listsFrom y)) ]
```

- Ein `QuickCheck`-Generator entspricht einer Wahrscheinlichkeitsverteilung über einer Menge, nicht der Menge selbst
- Obige Gewichtszuordnung führt dazu, dass Listen mit durchschnittlicher Länge 4 erzeugt werden

Die Klasse Arbitrary

Werden Nichtstandardgeneratoren wie `orderedLists` häufig verwandt, empfiehlt sich, diesen Typ zu einer Instanz von `Arbitrary` zu machen:

```
newtype OrderedList a = OL [a]

instance (Num a, Arbitrary a) => Arbitrary(OrderedList a) where
  arbitrary = liftM OL orderedLists
```

Die Redefinition von `insert`

```
insert :: Ord a => a -> OrderedList a -> OrderedList a
```

erlaubt dann die automatische Argumentgenerierung.

Kontrolle der Größe erzeugter Testfälle

- Für typbasierte Testfallgenerierung oft sinnvoll
- In `QuickCheck` explizit unterstützt

Und zwar mittels:

```
sized :: (Int -> Gen a) -> Gen a  -- Zur Def. groessenabh. Gen.

sized $ \n -> do len <- choose (0,n) -- Anwendung von sized
                 vector len          -- in der Def. des Standard-
                                     -- listengenerators

vector n = sequence [arbitrary | i <- [1..n]] -- erzeugt Zufallsliste
                                              -- der Laenge n

resize :: Int -> Gen a -> Gen a -- zur Kontroll der Groesse
                                -- erzeugter Werte

sized $ \n -> resize (round (sqrt (fromInt n))) arbitrary
        -- Anwendung von resize
```


Generatoren für benutzerdefinierte Typen

Testfallgeneratoren für...

- *vordefinierte Typen*
  - werden von `QuickCheck` zur Verfügung gestellt
  - für *benutzerdefinierte Typen* ist dies nicht möglich
- *benutzerdefinierte Typen*
  - müssen vom Anwender durch Definition einer geeigneten Instanz der Klasse `Arbitrary` zur Verfügung gestellt werden
  - erfordern i.a., insbesondere im Falle rekursiver Typen, die Größe der erzeugten Testfälle zu kontrollieren

Beispiel: Binärbäume

Betrachte den Baumtyp `Tree`:

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

Folgende Festlegung des Testfallgenerators liegt nahe:

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


Beispiel: Binärbäume

Beachte:

- Die Gewichtszuordung (1 vs. 3) wurde mit dem Ziel gewählt, die Erzeugung allzuvieler trivialer Bäume der Größe 1 zu vermeiden
- *Problem*: Die Wahrscheinlichkeit, dass der Generator einen *endlichen* Baum erzeugt, liegt nur bei einem Drittel

  $\leadsto$ ...beachte dazu, dass Terminierung nur möglich ist, wenn alle erzeugten Teilbäume endlich sind. Mit wachsender Baumbreite ist diese Forderung, d.h. das immer der "terminierende" Zweig ausgewählt wird, an immer mehr Stellen gleichzeitig zu erfüllen

Beispiel: Binärbäume

Abhilfe:

- Verwendung des Parameters `size`, um
  - Terminierung und
  - "vernünftige" Größe

  der erzeugten Bäume sicherzustellen

Beispiel: Binärbäume

Umsetzung:

```
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)
```


Beispiel: Binärbäume

Bemerkung:

- `shrub` ist ein *Generator* für "kleinere" Bäume
- `shrub` ist nicht an einen speziellen Baum gefunden; die beiden Vorkommen von `shrub` erzeugen i.a. verschiedene Bäume
- Da die Größenbeschränkung für Teilbäume halbiert wird, ist die Gesamtgröße durch den Parameter `size` beschränkt
- Generatordefinitionen für rekursive Typen erfordern i.a. stets ein Vorgehen wie in diesem Beispiel

Testfallüberwachung

In der Praxis ist es sinnvoll...

- die erzeugten Testfälle zu überwachen
- und so einen Hinweis auf die Güte und Testfallüberdeckung eines `QuickCheck`-Laufs zu erhalten

`QuickCheck` stellt dafür...

- eine Reihe von Überwachungsmöglichkeiten zur Verfügung

Testfallüberwachung

Warum ist Testfallüberwachung sinnvoll?

Betrachte dazu wieder das Beispiel sortierten Einfügens in Listen:

```
prop_InsertOrdered :: Integer -> [Integer] -> Property
prop_InsertOrdered x xs = ordered xs ==> ordered (insert x xs)
```


Testfallüberwachung

Die Überprüfung von `prop_InsertOrdered` erfolgt von `QuickCheck` derart, dass...

- zufällig Listen generiert werden
- jede generierte Liste darauf geprüft wird, ob sie sortiert ist (herangezogener Testfall) oder nicht (verworfener Testfall)

Offenbar gilt...

- eine zufällig generierte Liste ist um so wahrscheinlicher sortiert je kürzer sie ist

Damit besteht die Gefahr, dass...

- die Eigenschaft `prop_InsertOrdered` i.w. mit ein- und zweielementigen Listen getestet wird
- auch ein erfolgreicher Test wenig aussagekräftig ist

Testfallüberwachung

Zur Überwachung stellt `QuickCheck` zur Verfügung einen...

- Kombinator `trivial`, wobei die Bedeutung von "trivial" benutzerdefinierbar ist

Beispiel:

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

mit

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial)
```


Testfallüberwachung

Beobachtung:

- 91% sind zu viele triviale Testfälle, um hinreichende Aussagekraft des Gesamttests sicherzustellen
- Der Operator `==>` in Testfallgeneratoren ist nur mit Bedacht zu benutzen

Abhilfe:

- Benutzerdefinierte Generatoren

  $\leadsto$ ...wie im Beispiel von `prop_InsertOrdered` im Stil von Folie 14

Testfallüberwachung

Der Kombinator `trivial` ist...

- Ausprägung eines allgemeineren Kombinators `classify`

  ```
  trivial p = classify p "trivial"
  ```


Testfallüberwachung

Mehrfache Anwendung von `classify` erlaubt eine verfeinerte Testfallüberwachung:

```
prop_InsertOrdered x xs = ordered xs =>
   classify (null xs) "empty lists" $
      classify (length xs == 1) "unit lists" $
         ordered (insert x xs)
```

Liefert:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
```


Testfallüberwachung

Der Kombinator `collect` erlaubt weitergehend auch über alle Testfälle Aufzeichnungen zu führen:

```
prop_InsertOrdered x xs = ordered xs =>
   collect (length xs) $ ordered (insert x xs)
```

Liefert:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
46% 0.
34% 1.
15% 2.
5% 3.
```


Fazit 1(3)

Generell gilt:

- Die Angabe von Spezifikationen ist wertvoll (auch ohne anschließenden formalen Korrektheitsbeweis)

Die Erfahrung zeigt:

- Auch die Angabe von Spezifikationen ist (zunächst) oft selbst fehlerbehaftet

Fazit 2(3)

`QuickCheck` ist ein effektives Werkzeug...

- Fehler in
  - Programmen und
  - Spezifikationen

  mit vergleichsweise geringem Aufwand aufzudecken.
- Testkosten
  - zu reduzieren
  - trotz gleichzeitig erhöhten Testumfangs

Fazit 3(3)

Untersuchungen von Richard Hamlet in...

Richard Hamlet. *Random Testing*. In J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 970-978, 1994

legen nahe, dass

- eine hohe Testanzahl auch bei *Random Testing* zu Ergebnissen hoher Aussagekraft führt

Prinzipiell gilt:

- Die Generierung zufälliger Tests ist "billig"

Somit spricht viel für...

- die regelmäßige Verwendung eines Werkzeugs wie `QuickCheck`!

Nächster Vorlesungstermin

- Mi, 21.06.2006, abweichend 15 - 17 Uhr, CompLang-Bibliothek