QuickCheck: Generierung von Testdaten für funktionale Programme

Fabian Reck

Betreuer:
Prof. Dr. Michael Hanus

(06.11.2006)

Inhalt

1 Einleitung

Obwohl allgemein bekannt ist, dass das Testen von Softwaresystemen nur die Anwesenheit von Fehlern zeigen und keine Fehlerfreiheit garantieren kann, ist und bleibt das Testen die wichtigste Methode zur Validierung und Verifikation von Software. Unterschiedlichen Quellen zufolge macht das Testen eines komplexen Softwaresystems im Durchschnitt zwischen 40% und 50% der gesamten Entwicklungskosten eines Softwareprojektes aus. Dieser hohe Aufwand motiviert die Entwicklung von automatischen Testsystemen, die helfen sollen Tests mit weniger Aufwand durchzuführen oder mit dem gleichen Aufwand genauer zu testen. Außerdem ermöglichen diese automatischen Tests die Software auch nach kleineren Änderungen zu Testen und damit Fehler schon frühzeitig zu finden, die zu einem späteren Zeitpunkt zu erheblichen Problemen führen könnten.

Ein Tool, das solche automatischen Tests ermöglicht, ist QuickCheck. QuickCheck wurde entwickelt, um Haskell Programme mit automatisch generierten Eingabedaten zu testen. Da QuickCheck *lightweight*, also komplett in Haskell geschrieben ist, beschränkt sich der Installationsaufwand auf das Importieren des Moduls QuickCheck in das zu testende Programm. Dies hat den weiteren Vorteil, dass der Entwickler, um QuickCheck verwenden zu können keine zusätzliche Sprache zu erlernen braucht. Alles was nötig ist, um ein Programm mit QuickCheck zu testen, kann in Haskell ausgedrückt werde. Der Entwickler muss dazu nur einige vom Modul bereitgestellte Funktionen kennen.

Um angeben zu können, was getestet werden soll und ob ein Test erfolgreich war oder nicht, nutzt QuickCheck eine in Haskell eingebettete Sprache zur Formulierung einer formalen Spezifikation. Diese Spezifikation, im Grunde eine spezielle Gruppe von Haskell Funktionen, die hier Properties genannt werden, kann direkt in die zu testenden Module geschrieben werden. Eine formale Spezifikation zum Testen zu benutzen soll verschiedenen Zwecken dienen. Zum Einen soll auf diese Weise die Motivation, schon frühzeitig im Entwicklungsprozess eine formale Spezifikation zu erstellen, erhöht und damit gleichzeitig das Bewusstsein für die genauen Anforderungen an das Programm gestärkt werden. Zum Anderen kann so nicht nur das Programm gegenüber der Spezifikation getestet werden, sondern auch umgekehrt. So kann es durchaus vorkommen, dass beim Testen ein Fehler in der Spezifikation offenbart wird.

Diese Ausarbeitung soll einen Überblick über die Syntax von QuickCheck und die Funktionsweise von ausgewählten Teilen von QuickCheck geben. Hauptsächlich stützt sie sich auf ein Paper [1] der Entwickler von QuickCheck, Koen Claessen und John Hughes.

2 Grundlagen

2.1 Spezifikationen

Mit Hilfe der Syntax von QuickCheck lassen sich verschiedene etablierte Formalismen zur Spezifizierung von Programmen ausdrücken. Einige Beispiele, wie dies aussehen kann, finden sich in [2]. Drei dieser Formalismen sollen hier kurz vorgestellt werden.

2.1.1 Algebraische Spezifikation

Algebraische Spezifikationen dienen vor allem zur formalen Spezifikation von komplexen Datentypen. Die Syntax wird durch die Signatur der Operationen auf den Datentypen festgelegt, die Semantik durch Axiome. Wünschenswert ist dabei, dass die Axiome das Verhalten des Datentyps eindeutig festlegen.

Ein Beispiel hierfür ist eine Spezifikation der Haskell – Funktion `reverse. Reverse` erhält als Argument eine Liste und gibt eine Liste mit den selben Elementen in umgekehrter Reihenfolge zurück. Durch die folgenden zwei Axiome wird das Verhalten von `reverse` eindeutig festgelegt:

```
reverse[x] = [x]
reverse (xs++ys) = reverse ys ++ reverse xs
```

Das erste Axiom besagt hierbei, das `reverse` angewandt auf eine einelementige Liste keine Auswirkungen auf die Liste hat. Das zweite, dass reverse angewandt auf die Konkatenation von zwei Listen das gleiche bewirkt, wie die umgekehrte Konkatenation der Teillisten, auf die `reverse` angewandt wurde.

2.1.2 Modellbasierte Spezifikation

Die modellbasierte Spezifikation beschreibt ein System als mathematisches Modell des Systemzustandes. Dieses Modell wird zum Beispiel aus bekannten mathematischen Konstrukten wie Mengen und Funktionen aufgebaut. Operationen auf diesem Modell sind dann durch ihre Wirkung auf den Zustand des Modells definiert.

Definiert man eine Funktion, die den aktuellen Systemzustand in einen Zustand des Modells überführt, so lässt sich das System gegenüber dem Modell testen, indem auf beide die gleichen Operationen angewandt werden und dann überprüft wird, ob sich System und Modell in einem äquivalenten Zustand befinden. Abbildung 1 Zeigt diesen Vorgang.

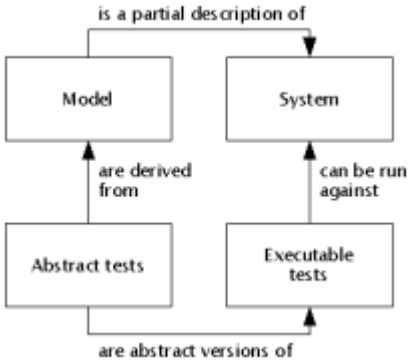

Abbildung 1: Darstellung des Testablaufs mit einer Modellbasierten Spezifikation

In [2] wird auf diese Weise eine Queue modellhaft als Liste spezifiziert und gegen eine imperative Implementierung getestet.

2.1.3 Spezifikation durch Vor- und Nachbedingung

Durch das explizite Angeben von Vor- und Nachbedingungen zu Programmen oder Programmteilen lassen sich bekanntermaßen Korrektheitsbeweise durchführen. Auch Spezifikationen lassen sich durch das so genannte Hoaretripel (`{p}e{q}`) notieren. Dabei sind `p` und `q` boolesche Ausdrücke und `e` ein Programm oder ein Programmteil. Das ganze liest sich dann folgendermaßen: Wenn vor der Ausführung von `e` der Ausdruck `p` wahr ist, dann muss nach der Ausführung von `e q` gelten.

Beispielsweise ist

```
{ x + 1 > a }
x := x + 1;
{ x > a }
```

eine solches Hoaretripel. In QuickCheck lassen sich solche Aussagen im Allgemeinen nur in monadischen Berechnungen testen, denn wie die Worte Vor- und Nachbedingung ja schon vermuten lassen spielt die Reihenfolge der Ausführung eine Rolle.

2.2 Monaden

Da Haskell eine reine funktionale Sprache ist, hängt das Ergebnis einer Berechnung nur von den

Parametern ab, die explizit übergeben wurden. Insbesondere spielt die Ausführungsreihenfolge beim Ergebnis keine Rolle. Mit Hilfe von Monaden lässt sich die Ausführungsreihenfolge jedoch festlegen. Am Beispiel der Ein- und Ausgabe lässt sich am einfachsten erkennen, warum es nötig ist die Reihenfolge der Programmausführung kontrollieren zu können.

Angenommen es gäbe eine Funktion `readInt`, die einen Integerwert von außen, zum Beispiel von der Tastatur, einliest und zurück gibt. Wenn wir dann eine Funktion `subInput` folgendermaßen definieren,

```
subInput = readInt - readInt
```

könnte diese Funktion, wegen fehlender Kontrolle über die Auswertungsreihenfolge, bei identischen Eingaben prinzipiell auf zwei verschiedene Ergebnisse kommen.

In Haskell ist eine Monade ein Datentyp, für den die beiden Operationen *bind* `(>>=)` und *return* `(return)` mit folgenden Signaturen definiert sind.

```
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
```

Dabei sind `a` und `b` normale Typvariablen und `m` eine Variable für den Typkonstruktor einer Monade. Der Operator `return` überführt eine normale Berechnung in eine monadische und bewirkt ansonsten gar nichts. Der Operator *bind* dagegen ermöglicht nun tatsächlich die Kontrolle über die Ausführungsreihenfolge. *Bind* führt zuerst eine monadische Berechnung aus und gibt dann das Ergebnis an eine weitere monadische Berechnung weiter.

In Haskell ist die IO – Monade zur Durchführung von Ein- und Ausgabeoperationen gedacht. Die oben verwendete Funktion `readInt` muss dann

```
readInt :: IO Int
```

sein. Nun können wir tatsächlich die Funktion `subInput` so definieren, dass sie ein deterministisches Verhalten zeigt:

```
subInput :: IO Int
subInput =
     readInt >>= (\a -> readInt >>= (\b -> return (a - b)))
```

Die Funktion `subInput` liest zuerst eine Zahl von der Tastatur ein und gibt diese dann nach der Definition des *bind* – Operators an eine Funktion weiter. Diese Funktion, hier in der Lambda – Notation, liest eine weitere Zahl von der Tastatur ein und übergibt diese wiederum mithilfe des *bind* – Operators an eine weitere Funktion, die nun die Subtraktion durchführt. Damit das Ergebnis, wie von der Signatur des *bind* – Operators verlangt, innerhalb der Monade bleibt, wird `return` verwendet.

Im Allgemeinen gibt es keine Möglichkeit eine monadische Berechnung wieder zu verlassen. Insbesondere bei der IO – Monade würde dies sogar zu den selben Problemen führen, die Monaden notwendig gemacht haben. Bei anderen Monaden kann es dagegen durchaus möglich und sinnvoll sein, ein Ergebnis aus dem Kontext einer monadischen Berechnung wieder herausholen zu können.

Mehr über Monaden findet sich in [3].

3 Hauptteil

QuickCheck ermöglicht das Testen von Programmen und Programmteilen, durch das Angeben von ausführbaren Spezifikationen bzw. Properties. Properties werden dabei direkt im zu testenden Haskell – Programm angegeben. Der Aufruf des Programms QuickCheck überprüft diese Properties dann mit einer festgelegten Anzahl, standardmäßig 100, von zufällig generierten Testdaten.

3.1 Syntax von Properties

Properties sind im Prinzip normale Haskell-Funktionen deren Resultat den Typ `Bool` hat. Die Namen der Properties sollten mit „`prop_`“ beginnen, damit das mitgelieferte Script Properties von anderen Funktionen unterscheiden und automatisch testen kann.

3.1.1 Ein erstes Beispiel

Als erstes soll hier die schon aus dem Grundlagenteil bekannte Funktion `reverse` getestet werden. Außer den beiden schon bekannten Axiomen

```
reverse [x] = [x]
```

und

```
reverse (xs++ys) = reverse ys ++ reverse xs
```

soll allerdings noch eine dritte Eigenschaft getestet werden:

```
reverse (reverse xs) = xs
```

Doppeltes Umkehren der Reihenfolge verändert die Liste nicht.

Das Formulieren der entsprechenden QuickCheck-Properties beschränkt sich nun auf das Finden von passenden Funktionsnamen und das Angeben einer Signatur.

```
prop_RevUnit :: Int -> Bool
prop_RevUnit x =
     reverse [x]  == [x]


prop_RevApp :: [Int] -> [Int] -> Bool
prop_RevApp xs ys =
      reverse (xs++ys) == reverse ys ++ reverse xs


prop_RevRev :: [Int] -> Bool
prop_RevRev xs =
     reverse (reverse xs) == xs
```

Auch wenn diese Properties eigentlich polymorph sind, benötigt QuickCheck einen expliziten Typ

mit dem es diese dann testen kann.

3.1.2 Testdaten mit Vorbedingung

Manche Properties gelten nur unter bestimmten Bedingungen. Ein Property, das testen soll, ob die Funktion `max` das richtige Argument zurück gibt, kann wie folgt formuliert werden:

```
prop_MaxLe :: Int -> Int -> Bool
prop_MaxLe x y = x>y || max x y == y
```

In diesem Fall wären etliche Tests nur deshalb erfolgreich, weil die Vorbedingung nicht erfüllt ist. Um ausschließlich mit Daten testen zu können, die die Vorbedingung erfüllen, stellt QuickCheck einen speziellen Kombinator bereit, mit dem das Property dann wie folgt aussieht:

```
prop_MaxLe :: Int -> Int -> Property
prop_MaxLe x y = x<= y ==> max x y == y
```

Der Ergebnistyp eines solchen Properties ist `Property`. Dies ermöglicht QuickCheck zu erkennen, ob die Vorbedingung bei einem Test erfüllt war.

Um Endlosschleifen bei nicht erfüllbaren Vorbedingungen zu vermeiden, begrenzt QuickCheck die generierten Testdaten auf 1000. Bei einigen Vorbedingungen ist es allerdings unwahrscheinlich, dass unter den 1000 erzeugten Testdaten 100 die Bedingung erfüllen. In diesen Fällen wird die Anzahl der erfolgreichen Tests ausgegeben. Alternativ lassen sich auch benutzerdefinierte Testdatengeneratoren angeben, auf die später eingegangen wird.

3.1.3 Überwachung der Testdaten

Um beurteilen zu können, wie aussagekräftig ein Testlauf tatsächlich ist, ist es notwendig die Verteilung der Testdaten über der Menge der möglichen Eingaben zu kennen. Hier kommen die von QuickCheck bereitgestellten Kombinatoren `classify` und `collect` zum Einsatz.

`Classify` ermöglicht es, Testdaten, die einer angegebenen Bedingung genügen, einer Klasse zuzuordnen. Die Anwendung wird an einem Property gezeigt, das das Einfügen in eine geordnete Liste testet:

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
     ordered xs ==>
          classify (null xs) "trivial" $
          ordered (insert x xs)
```

Die Funktion `classify` fügt dem Ergebnistyp `Property` sozusagen ein Label hinzu, falls das erste Argument zu wahr ausgewertet wird. Andernfalls passiert gar nichts.

Ein Aufruf dieses Properties mit QuickCheck könnte folgende Ausgabe produzieren:

```
OK, passed 100 tests (43% trivial).
```

Das Ergebnis, dass sich ein großer Teil der Tests auf das Einfügen in die leere Liste beschränkt, verwundert nicht, wenn man bedenkt, dass die Wahrscheinlichkeit, dass eine zufällig erzeugte Liste geordnet ist, mit zunehmender Größe deutlich sinkt.

Das Ganze wird durch den Einsatz des Kombinators `collect` noch deutlicher. `Collect` ordnet jedem Testfall einen Wert zu und gibt dann die Verteilung dieser Werte unter den Testdaten aus.

```
prop_Insert :: Int -> [Int] -> Poperty
prop_Inseert x xs =
     ordered xs ==>
          collect (length xs) $
               ordered (insert x xs)
```

Die Funktion `collect` funktioniert im Grunde genau wie `classify`, mit dem Unterschied, dass `collect` jedem Ergebnis ein Label zuordnet.

Ein mögliches Resultat dieses Properties wäre:

```
OK, passed 100 tests.
49% 0.
32% 1.
12% 2.
4% 3.
2% 4.
1% 5.
```

Um die Verteilung der Testdaten besser kontrollieren zu können, ermöglicht QuickCheck den Einsatz von benutzerdefinierten Testdatengeneratoren. Diese Generatoren lassen sich direkt in den Propertys benutzen, wie das folgende Beispiel zeigt:

```
prop_Insert :: Int -> Property
prop_Insert x =
     forAll orderedList $ \xs ->
          ordered (insert x xs)
```

Die Funktion `forAll` erzeugt dabei mit Hilfe des ihr übergebenen Generators einen Wert und übergibt diesen an ihr zweites Argument. Das zweite Argument muss dabei eine testbare Funktion sein, das heißt, ihr Resultat muss vom Typ `Bool` oder `Property` sein.

Im Gegensatz zu der vorherigen Version von `prop_Insert` hat diese Version dann nur noch ein Argument, für das QuickCheck dann den Standardgenerator verwendet.

3.2 Testdatengeneratoren definieren

3.2.1 Die Klasse Arbitrary

Um zu definieren, für welche Typen Testdaten erzeugt werden können, wurde die Typklasse `Arbitrary` eingeführt.

```
class Arbitrary a where
     arbitrary :: Gen a
```

Datentypen, für die Testdaten generiert werden sollen müssen also die Funktion `arbitrary` implementieren. Diese gibt einen Generator `Gen` zurück, der einen Generator für den gewünschten Typ `a` repräsentiert. `Gen` ist dabei wie folgt definiert:

```
newtype Gen a = Gen (Int -> Rand -> a)
```

`Gen` kapselt also eine Funktion, die ein `a` erzeugen kann. Das erste Argument vom Typ `Int` ist dabei eine positive Schranke für die Größe der erzeugten Testdaten, das zweite Argument ist ein Anfangswert für einen Zufallsgenerator, mit dessen Hilfe die Funktion dann pseudozufällige Werte

produzieren soll.

Um komplexere Testdatengeneratoren aus einfacheren zu erzeugen, wird zuerst ein einfacher Generator definiert.

```
choose :: (Int, Int) -> Gen Int
choose bounds = Gen (\n r -> fst (randomR bounds r))
```

Die Haskell-Funktion `randomR` erzeugt dabei ein Tupel bestehend aus einer Zufallszahl in den angegebenen Grenzen und einem neuen, unabhängigen Anfangswert für einen Zufallsgenerator. Mit `fst` wird dann der Zufallswert aus dem Tupel extrahiert. Der Größenparameter `n` wird bei diesem Generator ignoriert.

Um Testdatengeneratoren kombinieren zu können wird `Gen` zu einer Instanz der Haskell – Klasse `Monad` gemacht. Dazu wurden die folgenden beiden Methoden für die Klasse `Gen` implementiert:

```
return :: a -> Gen a
return a = Gen (\n r -> a)
```

`Return` erzeugt also einen Generator, der unabhängig von den Argumenten eine Konstante zurück gibt.

```
(>>=) :: Gen a -> (a -> Gen b) -> Gen b
Gen m1 >>= k =
     Gen (\n r0 -> let (r1, r2) = split r0
                           Gen m2 = k (m1 n r1)
                   in m2 n r2)
```

Der *bind* – Operator verhält sich nun so, wie erwartet. Der übergebene Generator generiert zuerst einen Wert vom Typ `a` und wendet darauf die Funktion `k` an, die einen Generator für den Typ `b` erzeugt. Damit die beiden Generatoren unabhängige Ergebnisse produzieren, wird die Funktion `split` verwendet, die aus einem Zufallswert zwei unabhängige Werte erzeugt.

Mit diesen Operatoren lassen sich nun Generatoren für viele Typen definieren, als Beispiele werden hier Testdatengeneratoren für Integer und Paare angegebenen:

```
instance Arbitrary Int where
     arbitrary = choose (-20, 20)


instance  (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
     arbitrary = liftM2 (,) arbitrary arbitrary
```

Im zweiten Beispiel müssen die Typen `a` und `b` Instanzen der Klasse Arbitrary sein. Die Haskell – Funktion `liftm2` wendet den Operator zum Bilden von Paaren `(,)` an, um aus den Generatoren für die Typen `a` und `b` einen Generator für den Typ `(a,b)` zu erzeugen.

3.2.2 Erzeugen von Generatoren

Das Erzeugen von Generatoren für benutzerdefinierte Typen wird dem Benutzer überlassen. Dadurch hat der Benutzer die Kontrolle über die Verteilung der Testdaten. Um Generatoren erzeugen zu können, stellt QuickCheck eine Reihe von Kombinatoren bereit.

Der einfachste dieser Kombinatoren ist `oneof`. Dieser Kombinator erhält als Argument eine Liste von Testdatengeneratoren und wählt einen davon aus. Jeder Generator wird dabei mit gleicher Wahrscheinlichkeit gewählt.

Sei beispielsweise der Datentyp `Colour` wie folgt definiert:

```
data Colour = Red | Blue | Green
```

Dann lässt sich `oneof` nutzen, um einen Generator zu erzeugen, der zufällig einen der drei Werte liefert.

```
instance Arbitrary Colour where
     arbitrary = oneof [return Red, return Blue, return Green]
```

Wie bereits gesagt liefert `return` einen Generator, der unabhängig von seinen Argumenten immer den selben Wert zurückgibt.

Unter Umständen kann es sinnvoll sein, bei der zufälligen Wahl der Generatoren eine andere Verteilung als die Gleichverteilung zu verwenden. Dazu kann der Kombinator `frequency` verwendet werden. Dieser wählt aus einer Liste von Paaren, die aus einem Generator und einer Gewichtung bestehen, mit einer Verteilung, die der Gewichtung entspricht, zufällig einen Generator aus.

Zum Beispiel kann `frequency` beim Erzeugen eines Generators für Listen verwendet werden.

```
instance Arbitrary a => Arbitrary [a] where
     arbitrary = frequency
       [(1, return []), (4, liftM2 (:) arbitrary arbitrary)]
```

Hierbei wird eine Liste erzeugt, indem rekursiv zufällige Elemente zu einer bereits erzeugten, zufälligen Liste hinzugefügt werden. Durch Berücksichtigung der Gewichtung werden im Schnitt Listen der Länge vier erzeugt.

Nicht immer reicht `frequency` aus, um die Größe der generierten Testdaten zu kontrollieren. Zu diesem Zweck wurde bei der bereits vorgestellten Definition des Datentyps `Gen` bereits ein Parameter zur Beschränkung der Größe vorgesehen. Eine Implementierung eines Generators für Binärbäume, ohne Berücksichtigung dieses Parameters, macht deutlich, warum dieser nötig ist.

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

instance Arbitrary a => Arbitrary (Tree a) where
     arbitrary = frequency
       [(1, liftM Leaf arbitrary)
        , (2, liftM2 Branch arbitrary arbitrary)]
```

Damit die Bäume nicht zu klein werden, wird hier `frequency` benutzt. Das Problem ist nun, dass durch die beiden rekursiven Aufrufe des Generators beim Erzeugen eines `Branch` die Wahrscheinlichkeit, dass der Generator terminiert nur bei 50% liegt. Außerdem können die erzeugten Testdaten sehr groß werden, so dass das Testen unter Umständen sehr lange dauert. Eine modifizierte Version des Generators für Binärbäume soll nun die Größenbeschränkung berücksichtigen.

Um Zugriff auf den Wert der Beschränkung zu erhalten ist ein weiterer Kombinator nötig.

```
sized :: (Int -> Gen a) -> Gen a
```

Dieser Kombinator gibt die aktuelle Größenbeschränkung an sein Argument, eine Funktion, die eine Zahl erwartet und einen Generator zurück liefert, weiter und erzeugt so einen Generator mit Größenbeschränkung. Die neue Version des Binärbaumgenerators sieht dann folgendermaßen aus:

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

```
arbTree 0 = liftM Leaf arbitrary

arbTree n = frequency
     [(1, liftM Leaf arbitrary)
       , (4 liftM2 Branch (arbTree (n 'div' 2))
                          (arbTree (n 'div' 2))) ]
```

Auf diese Weise ist die Anzahl von `Branch` im Baum durch den Größenparameter beschränkt, der durch die Funktion `sized` an `arbTree` weitergegeben wird. An diesem Beispiel lässt sich auch erkennen, dass die Interpretation der Größenbeschränkung ausschließlich dem Benutzer überlassen ist. Denkbar wäre zum Beispiel auch gewesen, die Größenbeschränkung als Beschränkung der Tiefe des Baumes zu interpretieren.

Während eines Testlaufs verändert QuickCheck die Größenbeschränkung. Begonnen wird mit einer kleinen Schranke, die dann immer weiter erhöht wird. Auf diese Weise soll sowohl die Wahrscheinlichkeit, ein möglichst kleines Gegenbeispiel zu finden erhöht, als auch eine möglichst große Vielfalt an Testdaten erreicht werden.

3.2.3 Generatoren für Funktionen

Um Properties überprüfen zu können, die Funktionen als Argumente erwarten, werden Generatoren benötigt, die zufällige Funktionen erzeugen können. Um zu verstehen, wie ein solcher Generator erzeugt werden kann, hilft es, sich vor Augen zu führen, dass ein Generator für den Funktionstyp `(a->b)` eine Funktion mit dem Typ `Int -> Rand -> a -> b` ist. Durch Veränderung der Reihenfolge der Argumente kann eine äquivalente Funktion vom Typ `a -> Int -> Rand -> b` definiert werden, die damit ja den `Typ a -> (Gen b)` hat.

Als Zwischenschritt auf dem Weg zu einem Generator für `(a -> b)` wird die Funktion

```
promote :: (a -> Gen b) -> Gen (a -> b)
```

definiert. Diese soll aus einer Funktion, die aus einem Wert vom Typ `a` einen Generator für den Typ `b` macht, einen Generator für die gewünscht Funktion erstellen. Promote ist folgendermaßen implementiert:

```
promote f = Gen (\n r -> \a ->
     let Gen m = f a in m n r)
```

Die Funktion `promote` macht also eigentlich nichts weiter, als tatsächlich die Argumente der übergebenen Funktion so umzuordnen, dass das Ergebnis den gewünschten Typ erhält.

Nun wird also noch eine Funktion mit dem Typ `a -> Gen b` benötigt. Um diese später konstruieren zu können, wird eine neue Klasse eingeführt:

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

Die Funktion `coarbitrary` soll einen bereits vorhandenen Generator für den Typ `b` in einer Weise verändern, die von ihrem Argument vom Typ `a` abhängt. Das bedeutet, dass die Klasse `Coarbitrary` für denjenigen Typ implementiert werden muss, der als Argument für die zufällig zu erzeugende Funktion dienen soll. Soll zum Beispiel eine Funktion vom Typ `(Int -> Bool)` erzeugt werden, so muss die Funktion `coarbitrary` für den Typ `Int` implementiert werden.

Durch diese Klasse kann nun die Funktion `arbitrary`, die dann den eigentlichen Generator liefern soll, unter der Annahme definiert werden, dass die Funktion `coarbitrary` bereits implementiert ist:

```
instance (Coarbitrary a, Arbitrary b) =>
                    Arbitrary (a -> b) where
     arbitrary = promote (/a -> coarbitrary a arbitrary)
```

Der Lambda – Ausdruck, der der Funktion `promote` übergeben wird, ist nun genau die gesuchte Funktion, die aus einem Wert vom Typ `a` einen Generator für den Typ `b` erzeugt. Nach der Umordnung der Argumente durch `promote` erhalten wir also den Generator für zufällige Funktionen vom Typ `(a -> b)`.

Offen ist nun jedoch noch die Frage nach einer sinnvollen Implementierung von `coarbitrary`. Zu diesem Zweck wird zunächst eine Funktion `variant` definiert:

```
variant :: Int -> Gen a -> Gen a
variant v (Gen m) = Gen (\n r ->
     m n (rands r !! (v + 1)))
     where
        rands r0 = r1 : rands r2 where (r1, r2) = split r0
```

Diese Funktion verändert den ihr übergebenen Generator, indem sie aus dem eigentlich an den ursprünglichen Generator übergebenen Zufallswert eine Liste von pseudozufälligen Zahlen aufbaut. Der dadurch erzeugte Generator hängt nun insoweit von der an `variant` übergebenen Zahl `v` ab, als dem Generator nun, statt des ursprünglichen Zufallswerts, der Wert an der Stelle `v + 1` der Liste der pseudozufälligen Zahlen übergeben wird.

Tatsächlich wäre `variant` eine gültige Implementierung für `coarbitrary` für den Typ `Int`. Da die Laufzeit von `variant` aber exponentiell zur Länge des Integerwerts ist, wird wohl eine andere Implementierung verwendet.

Eine Eigenschaft von `variant` ist, dass für eine beliebige Liste von nicht negativen Integerwerten `[n1, n2, ... , nk]` die Hintereinanderausführung

```
variant n1 . variant n2 . ... . variant nk
```

eine Funktion liefert, die für verschiedene Listen von Integerwerten mit hoher Wahrscheinlichkeit einen Generator auf unabhängige Weise verändert.

Mit Hilfe von `variant` ist es nun recht einfach Instanzen von `coarbitrary` zu definieren. Für den Typ `Bool` wird `coarbitrary` folgendermaßen implementiert:

```
instance Coarbitrary Bool where
     coarbitrary b =
          if b then variant 0 else variant 1
```

Mit dieser Definition sind die durch `coarbitrary True g` und `coarbitrary False g` erzeugten Generatoren voneinander unabhängig.

Um die exponentielle Laufzeit zu umgehen, kann die Instanz von `Coarbtrary` für den Typ `Int` sein Argument in eine Liste von Bits zerlegen, die der Binärdarstellung des Integers entspricht und so unter Verwendung von `variant` einen gegebenen Generator verändern.

4 Anwendungsbeispiele

4.1 Größter gemeinsamer Teiler

Erfolgreich konnte QuickCheck bei dem Testen eines Programms eingesetzt werden, das den größten gemeinsamen Teiler zweier natürlicher Zahlen mit Hilfe des euklidischen Algorithmus

berechnet. Folgendes Programm wurde dabei getestet:

```
module GGT where

import QuickCheck

ggT :: Int -> Int -> Int
ggT a b = if (a < b) then ggT b a
                     else let x = a `mod` b in
                       if (x == 0) then b
                                   else ggT b x
```

Um das Programm testen zu können, wurden die folgenden drei Properties angegeben:

```
prop_comm_ggT :: Int -> Int -> Property
prop_comm_ggT a b = a > 0 && b > 0 ==>
                      ggT a b == ggT b a
```

Das erste Property sollte sicherstellen, dass die definierte Funktion kommutativ ist.

```
prop_div_ggT :: Int -> Int -> Property
prop_div_ggT a b = a > 0 && b > 0 ==>
              a `mod` ggT a b == 0 && b `mod` ggT a b == 0
```

Dieses Property überprüft, ob das Ergebnis der Funktion `ggT` auch tatsächlich ein Teiler der beiden Argumente ist.

```
prop_biggest_ggT :: Int -> Int -> Property
prop_biggest_ggT a b = a > 0 && b > 0 ==>
                ggT (a `div` ggT a b) (b `div` ggT a b) == 1
```

Das letzte Property soll überprüfen, ob die Funktion auch tatsächlich den größten gemeinsamen Teiler gefunden hat, indem getestet wird, ob ein weiterer Aufruf von `ggt` mit den ursprünglichen Argumenten geteilt durch den zuerst gefundenen größten gemeinsamen Teiler als Ergebnis 1 liefert.

Tatsächlich wurde beim Testen kein Fehler in der Funktion gefunden. Was wohl darauf zurückzuführen ist, dass die Funktion sehr kurz und damit mit großer Wahrscheinlichkeit korrekt ist.

An diesem Beispiel lässt sich jedoch gut erkennen, dass der Benutzer die Properties mit Bedacht wählen sollte. Denn obwohl die Erfüllung dieser drei Properties für das korrekte Funktionieren einer Implementierung des ggT – Algorithmus notwendig ist, ist sie noch nicht ausreichend, um ihn vollständig zu spezifizieren. Eine Implementierung, die konstant den Wert eins zurück liefert würde den Test fälschlicherweise bestehen. Um zumindest diesen Fall auszuschließen, kann eines der Properties so modifiziert werden, dass ausgegeben wird, in wie vielen Prozent der Fälle der größte gemeinsame Teiler 1 war. Das neue Property sieht dann folgendermaßen aus:

```
prop_biggest_ggT :: Int -> Int -> Property
prop_biggest_ggT a b = a > 0 && b > 0 ==>
             classify (ggT a b == 1) "ggT gleich 1" $
             ggT (a `div` ggT a b) (b `div` ggT a b) == 1
```

Ein Aufruf der Funktion quickCheck liefert nach nur ein bis zwei Sekunden das folgende Ergebnis:

```
GGT> quickCheck prop_biggest_ggT
OK, passed 100 tests (70% ggT gleich 1).
```

4.2 Transponieren einer Matrix

In diesem Beispiel soll vor allem die Möglichkeit, Generatoren für Benutzerdefinierte Datenstrukturen zu erzeugen, veranschaulicht werden. Getestet wird die in Haskell bereits definierte Funktion `transpose`, die eine Liste von Listen transponiert. Hier soll jedoch `transpose` nur auf vollständige `n x m` - Matrizen angewandt werden.

QuickCheck hat zwar bereits einen vordefinierten Generator für zweidimensionale Listen, diese entsprechen aber im Allgemeinen keinen Matrizen.

Aus diesem Grund wird zunächst ein Generator für vollständige Matrizen definiert.

```
arbCol :: Int -> Gen [Int]

arbCol (m+1) =
   liftM2 (:) arbitrary (arbCol m)

arbCol 0 = return []
```

Die Funktion `arbCol` erzeugt einen Generator, der Listen einer bestimmten Länge generiert, die mit zufälligen Integer – Werten gefüllt sind.

```
arbMatrix :: Int -> Int -> Gen [[Int]]

arbMatrix 0 _ = return []

arbMatrix (n+1) m = liftM2 (:) (arbCol m) (arbMatrix n m)
```

Nun wird `arbCol` genutzt, um mit der Funktion `arbMatrix` einen Generator für Matrizen mit bestimmten Dimensionen zu erzeugen.

```
fullMatrix :: Gen [[Int]]

fullMatrix = sized (\maxSize -> choose(1,maxSize + 1)
                 >>= (\n -> choose (1,maxSize + 1)
                       >>= (\m -> arbMatrix n m)))
```

Damit jedoch nicht nur die Werte in der Matrix zufällig sind, werden mit dem bereits bekannten Generator `choose` auch die Dimensionen der Matrix zufällig gewählt. Dabei wird der Größenparameter mit dem Kombinator `sized` ermittelt, damit die Matrizen in einem gewissen Rahmen bleiben.

Nun, da ein Generator für zufällige Matrizen definiert ist, können die zu testenden Properties definiert werden. Um das zu erleichtern wird die Funktion `sameLength` benutzt.

```
sameLength :: Int -> [[Int]] -> Bool
sameLength _ [] = True

sameLength len (x:xs) = if length x == len
```

```
                                    then sameLength len xs
                                    else False
```

Diese Funktion testet, ob alle in der Liste enthaltenen Listen die Länge `len` haben. Nun kann getestet werden, ob die vom Generator erzeugten Matrizen tatsächlich volle `n x m` Matrizen sind.

```
prop_wellFormed :: Property
prop_wellFormed = forAll fullMatrix
                  (\(x:xs) -> (sameLength (length x) xs))
```

Um den vorher definierten Generator für Matrizen zu benutzen, wird die Funktion `forAll` benutzt. Die damit erzeugten Listen werden dann daraufhin getestet, ob sie vollständigen Matrizen entsprechen.

Eine transponierte Matrix sollte immer noch eine vollständige Matrix sein, diese Eigenschaft wird vom folgenden Property getestet.

```
prop_wellFormed_trans :: Property
prop_wellFormed_trans = forAll fullMatrix
                  (\x -> let xs = transpose x in
                      sameLength(length (head xs)) (tail xs))
```

Nun soll noch getestet werden, ob eine Matrix von `transpose` auch tatsächlich transponiert wird. Dies soll getestet werden, indem vom Property `prop_isTransposed` überprüft wird, ob sich das Element, das sich an einer zufäligen Stelle (i, j) in der ursprünglichen Matrix befindet, nach dem Transponieren an der Stelle (j, i) ist.

```
prop_isTransposed :: Property
prop_isTransposed = forAll fullMatrix
         (\x -> forAll (choose(0,(length x) - 1))
            (\i -> forAll (choose(0,(length (head x)-1)))
              (\j -> (x!!i)!!j == ((transpose x)!!j)!!i)))
```

Da in diesem Beispiel eine schon in Haskell vordefinierte Funktion getestet wurde, war nicht zu erwarten, dass ein Fehler gefunden wird. Allerdings erzeugte die erste Version des Generators leere Matrizen, die dann beim Testen der Properties einen Fehler verursachten. Außerdem zeigte sich, dass die Interpretation des Größenparameters nicht unerheblich ist. Beim Testen der Properties wurden die erzeugten Matrizen teilweise so groß, dass der Speicher nicht ausreichte. In diesem Fall reichte es, den nutzbaren Speicher durch einen Parameter zu vergrößern. Denkbar wäre jedoch auch, die Ausdehnung der Matrizen in jeder Dimension durch die Wurzel des Größenparameters zu beschränken und nicht durch den Parameter selbst.

Als letztes zeigt dieses Beispiel deutlich, dass der Aufwand Generatoren und Properties zu definieren nicht unerheblich ist. Auch wenn hier eine vordefinierte Funktion getestet wurde, so ist der Aufwand die nötigen Funktionen für QuickCheck bereitzustellen auch bei selbst erstellten Programmen im Allgemeinen mindestens so groß, wie der Aufwand das Programm selbst zu schreiben.

5 Zusammenfassung

QuickCheck ermöglicht auf recht einfache Art und Weise, Haskell – Programme mit zufällig

generierten Daten zu testen. Dabei hat der Benutzer ein großes Maß an Kontrolle über die Art und die Verteilung der generierten Daten. Die Möglichkeit, schon während der Programmentwicklung Tests durchführen zu können, motiviert eine formale Spezifikation für das Programm zu erstellen. Dies kann den Entwicklungsprozess zusätzlich verbessern und hilft dabei Spezifikation und Programm konsistent zu halten.

Einer der größten Vorteile von QuickCheck ist, dass es vollständig in die Sprache Haskell integriert ist und das Erlernen der Syntax daher minimalen Zeitaufwand benötigt.

Wie alle Testverfahren kann auch QuickCheck keinen Beweis für die Korrektheit eines Programms liefern. Da die Struktur des getesteten Programms nicht analysiert wird, gibt es keine Garantie, dass tatsächlich der gesamte Code getestet wurde. An dieser Stelle muss der Benutzer selbst darauf achten, dass die erzeugten Testdaten ausreichend breit gestreut sind und idealerweise auch Randfälle mit abdecken.

Mit der hier vorgestellten Version von QuickCheck ist es zwar möglich, aber sehr aufwändig monadische Programme zu testen. Um dies zu vereinfachen, haben die Autoren von QuickCheck in [2] eine Syntaxerweiterung vorgestellt, die das Testen von monadischem Code erleichtert.

Alles in allem ist der Einsatz von QuickCheck beim Entwickeln von Haskell Programmen durchaus zu empfehlen.

Literaturverzeichnis

1: Claessen Koen, Hughes John, QuickCheck: A Lightweight Tool for Random Testing of Haskell Programms, In Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN, Seiten 268-279, 2000
2: Claessen Koen, Hughes John, Testing Monadic Code with QuickCheck, In Proc. of Haskell Workshop, ACM SIGPLAN, Seiten 65-77, 2002
3: Niels Decker, Monads, 2002, http://www.fh-wedel.de/~si/seminare/ss02/Ausarbeitung/4.monads/monads.html