Testen von Clean-Programmen mit GAST

Arne Schipper

15. November 2006

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Software spielt heute in immer mehr Gebieten eine entscheidende Rolle. Häufig wird Software in sicherheitskritischen und lebenswichtigen Systemen eingesetzt. Fehler können dramatische Folgen haben, sowohl finanziell als auch für das Leben.

Die Entwicklung von Software hat sich auch aus diesem Grunde stark verändert. Heutige Software ist zu umfangreich und leider vielfach zu kompliziert, um sie manuell zu testen. Hinzu kommen nicht unerhebliche Kosten für die Durchführung verschiedener Tests. Zum Teil verschlingt die Testphase oft 50% eines Projektes.

Hier setzen Systeme an, die das Testen von bestimmter Software automatisch übernehmen. Je nach Art der Programmiersprache werden unterschiedliche Ansätze verfolgt. Für funktional-logische Sprache bieten sich Korrektheitsbeweise für bestimmte Eigenschaften an. Tests können hier einfach und elegant formuliert werden.

GAST - Generic Automatic Software Test-system[1] - ist solch ein Testsystem für die funktionale Sprache CLEAN, die eine ähnliche Syntax wie Haskell und Curry aufweist.

1.2 Testen von Software

Wenn man über das Testen von Software spricht, muss einem bewusst sein, welche Arten von Tests überhaupt gemeint sind. Mögliche Tests wären:

- Eignungstest - Passt die Software überhaupt zu dem gewünschten Anwendungsbereich? Tests hierfür können sehr unterschiedlich aussehen.
- Effizienz - Führt die Software die Aufgaben in angemessener Zeit durch?
- Folgt die Software der Spezifikation - Ist die Software korrekt umgesetzt?

Es gibt viele Möglichkeiten Software zu testen. Zum Teil sind Anforderungen und Überprüfungen nicht disjunkt formuliert, oder dieses ist gar nicht möglich. Man kann natürlich Beta-Versionen einer Software Testern zur Verfügung stellen und so herausfinden, wo Fehler in der Software liegen. Fehler im Programmcode werden damit aber nicht notwendigerweise gefunden. Umgekehrt kann man bei funktionalen Sprachen elegant feststellen, ob eine Software einer Spezifikation genügt. Über die Korrektheit der Spezifikation sagt das natürlich nichts aus. Die Autoren haben ein System geschrieben, welches Code der Sprache CLEAN, über Schnittstellen auch Fremdcode, in CLEAN selbst überprüfen kann. Das funktionales Testen von Programmcode wird in vier Schritte unterteilt.

1. Formulierung einer Eigenschaft, die getestet werden soll
2. Erstellung von Testdaten
3. Ausführung des Tests
4. Analyse des Ergebnisses

Hierbei übernimmt das GAST-System die letzten drei Schritte. Was genau getestet werden soll, müssen wir selbst angeben. Zum Beispiel könnten wir eine Funktion *add* im Programm testen wollen, die zwei Zahlen addiert. Die Formulierung der zu testenden Eigenschaft wäre also, ob die Funktion *add* dasselbe Ergebnis liefert, wie der Operator + in CLEAN. Nun würde GAST eine gewisse Anzahl an Zahlen generieren. Bei der Durchführung des Tests würde geprüft, ob sich die Ergebnisse von *add* und dem Operator + bei gleicher Eingabe gleichen. Das Ergebnis würde dann ausgegeben.

Ein vollautomatisches Testen ist also (natürlich) nicht möglich. Aber schon durch die Formulierung der Eigenschaften, die getestet werden sollen, wird die Dokumentation und Qualität der zu erstellenden Software gesteigert.

Die Eigenschaften folgen der Prädikatenlogik der ersten Stufe. Es steht dem Entwickler frei, ob er nur Funktionen oder Datentypen testet, oder ganze Stücke des Programms.

Bei der Erstellung der Testdaten verfolgt GAST einen besonderen Ansatz. Viele bestehende Testsysteme genieren Daten zufällig. Dabei kommt es natürlich vor, dass unsinnige oder doppelte Daten zum Testen benutzt werden, was einen längeren Testlauf zur Folge hat. Zudem muss beim Überprüfen von benutzerdefinierten Datentypen oder Funktionen auch stets der Datentyp mit angeben werden, damit überhaupt passende Daten erstellt werden können. Bei GAST werden hingegen nur sinnvolle und nicht doppelte Testdaten erzeugt. Auch die Datentypen müssen nicht explizit angegeben werden. Durch einen generischen Datenerzeuger kann das System die erforderlichen Datentypen selbst ableiten.

Bei der Ausführung wird einfach die Eigenschaft mit den erstellten Testdaten getestet. Dazu wird von GAST der vom CLEAN-Compiler compilierte Code benutzt, um die Eigenschaft mit den Testdaten zu testen. Vorteil ist, dass keine semantischen Missverständnisse auftreten können, außerdem bleibt das Testsystem so relativ einfach. Eine vereinfachte Version kommt mit nicht einmal 500 Zeilen Code aus.

2 Grundlagen

2.1 CLEAN

CLEAN[2] ist eine funktionale Programmiersprache, die von der Software Technology Research Group der University of Nijmegen entwickelt wird. Als funktionale Sprache bietet sie folgenden Vorteile:

- Pure, faule (lazy), funktionale Sprache höherer Ordnung
- Referenziell transparent, keine Seiteneffekte
- Streng getypt mit Typen höherer Ordnung, polymorphen Typen, algebraischen Typen, abstrakten Typen, ...
- Pattern Matching, Guards
- Modular
- und vielem mehr ;)

Wir werden einmal ein kurzes Beispiel betrachten, das einige der oben genannten Konzepte erhält und gleichzeitig eine kurze Einsicht in die Syntax gewährt. Hierzu betrachten wir die Potenzfunktion.

```
power :: Int Int -> Int
power x 0 = 1
power x n = x * power x (n-1)
```

Wie erwartet, wird in der ersten Zeile der Typ der Funktion definiert. Es handelt sich also um eine Funktion, die zwei Integer als Eingabe und einen als Ausgabe hat. Die beiden anderen Zeilen geben die Berechnungsvorschrift der Funktion an. Bei funktionalen Sprachen wird nur angegeben, wie es zu dem gewünschten Ergebnis kommt, die Berechnung an sich wird nicht ausgeschrieben. Weitere zu erkennende Features wäre Pattern Matching und Rekursion.

Polymorphismus bedeutet, dass der Typ für viele Funktionen frei gewählt werden kann. Somit kann eine Definition einer Funktion für mehrere Typen angewendet werden. Statt eines konkreten Typs benutzt man einen Platzhalter:

```
square :: t -> t
square x = x*x
```

Voraussetzung ist hierbei, dass der Operator `*` für den Typ `t` definiert ist

Ein Beispiel für eine Funktion höherer Ordnung ist `twice`.

```
twice :: (Int -> Int) Int -> Int
twice f x = f (f x)
```

Die Funktion erwartet als Argumente eine Funktion und einen Integer. Führen wir `twice square 2` für eine Quadratfunktion `square` aus (hier nur für Integer), so erhalten wir 16.

CLEAN-Programme sind modular aufgebaut. Man hat für ein Modul eine Definitionsdatei (`*.dcl`) und eine Implemenationsdatei `*.icl`. Schreibt man nur ein einfaches Testprogramm, so reicht eine Implemenationsdatei `start.icl` mit folgenden Anweisungen:

```
module test            // Name des Moduls

import StdEnv          // stellt Grundlegendes bereit wie z.B. *

Start :: String        // Einstiegsfunktion
Start = "Hallo Welt"
```

Kompiliert wird das Programm mittels `clm start -o start`. Danach kann man das Programm mit `.\start` wie üblich aufrufen.

Ein weiteres Feature von CLEAN ist generische Programmierung, die über eine Erweiterung bereit gestellt wird.

2.2 Generische Programmierung

Ziel der generischen Programmierung ist es, Funktionen und Klassen möglichst allgemein zu halten. Auf diese Weise können sie Eingaben von unterschiedlichen Typen verarbeiten. So müssen Funktionen nur einmal generisch definiert werden und können dann für alle Typen, zum Teil auch für selbst definierte, benutzt werden.

Hierbei gibt es unterschiedliche Konzepte in der Implementation in verschiedenen Programmiersprachen. Auch die Realisierung unterscheidet sich. Zum einen können zur Kompilierzeit generische Typen direkt in native Typen übersetzt werden, zum anderen werden spezielle Methoden benutzt, die auf alle Typen arbeiten können. Auch Kombinationen von beidem werden angewendet.

Bekannte Beispiele generischer Programmierung sind die Standardbibliothek in `C++`, wo mit Templates gearbeitet wird. Dieses Konzept hat jedoch Einschränkungen, besonders wenn mit benutzerdefinierten Strukturen gearbeitet werden soll. Das Konzept von CLEAN ist deutlich konsequenter ausgelegt.

2.3 Generische Programmierung in CLEAN

Für das GAST-System ist generische Programmierung wichtig, um auch für benutzerdefinierte Datentypen Testdaten erstellen zu können.

2.3.1 Generische Typen

Die generische Programmierung in CLEAN[3] basiert auf einer universellen Baum-Repräsentation von Datentypen, von denen alle Datentypen abgeleitet werden. Ein Baum besteht hierzu aus drei Elementen.

```
:: UNIT   = UNIT
:: EITHER = LEFT a | RIGHT B
:: PAIR   = PAIR a b
```

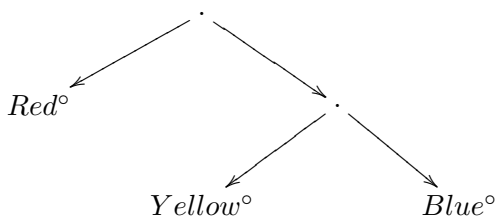
Mit diesen Elementen können wir nun jeden selbst definierten Datentyp erstellen. Sei dazu als Beispiel folgender Datentyp definiert:

```
:: Colour = Red | Yellow | Blue
```

Der zugehörige generische Datentyp ergibt sich daraus direkt aus oben genannten Regeln, er wird fortan mit einem ° von dem konkreten Datentypen unterschieden.

```
:: Colour° = EITHER UNIT (EITHER UNIT UNIT)
```

Daraus ergibt sich die Baumdarstellung,



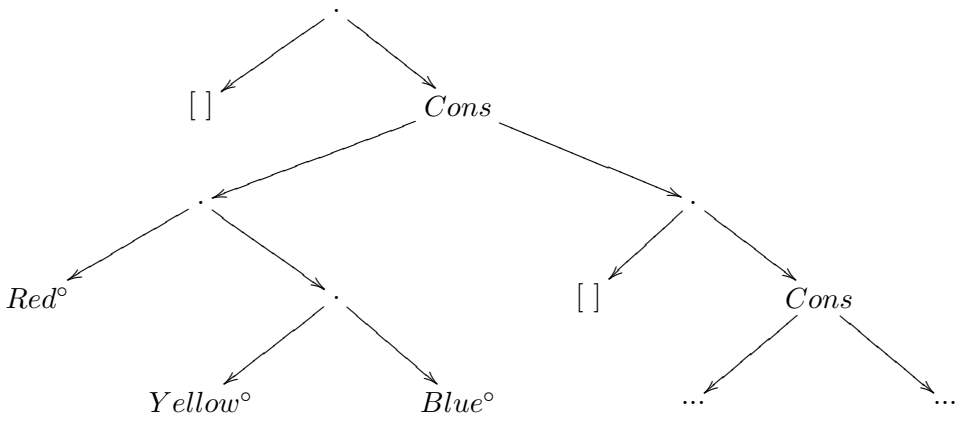
die auch gleich anschaulich die generischen Konstruktoren erklärt.

```
Red°    = LEFT UNIT
Yellow° = RIGHT (LEFT UNIT)
Blue°   = RIGHT (RIGHT UNIT)
```

Beachten muss man die Repräsentation von `[] = Nil`, welche mit `LEFT UNIT` dieselbe ist wie von z.B. `Red`.

Cons wird durch `PAIR` repräsentiert. Veranschaulicht wird das an der Konstruktion einer Liste, die alle Farben enthalten kann. (`= [ Color° ]°`). Dieses sieht als Baum

wie folgt aus:



2.3.2 Generische Funktionen

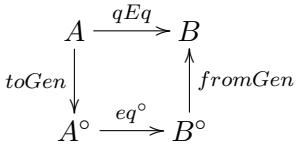
Da wir nun eine generische Repräsentation für Datentypen zu Verfügung haben, können wir auch Funktionen darauf generisch definieren. Da bedeutet, anstatt z.B. einer Vergleichsfunktion für jeden Datentyp wird eine generische Funktion für alle Datentypen definiert. Hierbei gibt es zwei Möglichkeiten. Zum einen kann man eine generische Funktion mittels `derive` ableiten. Hierbei spielt das generische Typsystem seine volle Stärke aus. Zum anderen kann man auch explizit vorgeben, wie eine Funktion mit einem bestimmten Datentyp zu verfahren hat.

Wir müssen also nur noch die generische Funktion auf den grundlegenden generischen Typen definieren. Wir betrachten zur Veranschaulichung im Folgenden die Gleichheitsfunktion. Die Gleichheit auf den generischen Typen wird wie folgt definiert:

```
generic qEq a :: a a -> Bool

qEq{|UNIT|}             _              _              = True
qEq{|PAIR|}   eqx eqy (PAIR x1 y1) (PAIR x2 y2) = eqx x1 x2 && eqy y1 y2
qEq{|EITHER|} eql eqr (LEFT x)     (LEFT y)     = eql x y
qEq{|EITHER|} eql eqr (RIGHT x)    (RIGHT y)    = eqr x y
qEq{|EITHER|} eql eqr e1           e2           = False
```

Jetzt ist es möglich, eine generische Gleichheitsfunktion für jeden Datentyp abzuleiten. Man kann fortan `gEq` für die Datentypen benutzen, für die man es abgeleitet hat. Nach oben gezeigter Vorschrift werden vom Compiler für jeden Typ zwei Funktionen definiert, `toGen` und `fromGen`. Nun kann man die generische Vergleichfunktion nutzen, das CLEAN-System wird je nach Typ der Argumente die passenden Konvertierungsfunktionen benutzen.



Auffällig ist hierbei noch die etwas umständliche Angabe `eqx`, `eqy`, `eql` und `eqr`, die angibt, wie genau z.B. die Elemente eines Paares verglichen werden sollen. Um nun zwei Elemente des Typs `PAIR` zu vergleichen, müsste man schreiben:

```
gEq eq eq Pair_1 Pair_2
```

Das `eq` als zweites und drittes Argument gäbe hier an, wie die einzelnen Elemente eines Paares verglichen würde. Diese `eq`-Funktion müsste für `PAIR` definiert sein. Vom Compiler wird eine Kurzbezeichnung zur Verfügung gestellt, die einen gleichen Aufruf von `gEq` für alle dafür abgeleiteten Datentypen zulässt. Wir können es als `gEq|*|` benutzen. Um die Sache weiter zu vereinfachen, können wir einen neuen Operator definieren, der der Vergleichsfunktion entspricht.

```
(===) infix 4 :: !a !a -> Bool | gEq{|*|} a
(===) x y = gEq{|*|} x y
```

Damit können wir nun `Typ_1 === Typ_1` für alle abgeleiten Typen schreiben. Der Zusatz `| gEq{|*|} a` hinter der Typdefinition der Funktion gibt dabei an, dass die Funktion `===` für alle Typen `a` definiert wird, für die eine Vergleichsfunktion `gEq{|*|}` existiert. Wollen wir für einen Typ keine abgeleitete Funktion benutzen, sondern eine eigene definieren, so geht dieses einfach mit

```
gEq{|INT|} x y = x==y
```

Die oben definierte Funktion `===` würde nun genau dasselbe leisten wie `==` für Integer.

3 GAST

Nachdem nun alle wichtigen Grundlagen bekannt sind - am wichtigsten war die generische Programmierung für CLEAN - kann nun das GAST-System vorgestellt werden. Wie eingangs erwähnt, konzentrieren sich die Autoren des GAST-System auf das funktionale Testen von Software. Dazu führen sie 4 Schritte ein, nämlich

- (a) das Formulieren von zu testenden Eigenschaften
- (b) das Generieren von Testdaten,
- (c) das Ausführen der Tests und schließlich
- (d) die Analyse der Ergebnisse.

Das CLEAN-System führt die letzten drei Schritte automatisch aus, so dass dem Entwickler das Formulieren der Eigenschaften überlassen bleibt, welches in CLEAN selbst erfolgt.

3.1 Eigenschaften

Das CLEAN-System kann Eigenschaften der Prädikatenlogik erster Stufe testen. Um dieses in CLEAN zu repräsentieren, werden die Eigenschaften als Funktionen geschrieben.

Im Folgenden wird vorausgesetzt, dass in CLEAN die Operatoren `||` und `&&` vorhanden sind, welche den logischen `oder` und `und` Operatoren entsprechen.

Um die Formulierung von Eigenschaften zu verstehen, betrachten wir ein kleines Beispiel. Dazu definieren wir eine Funktion `or`. Wir könnten dann als Eigenschaft fordern, dass unsere `or` Funktion dasselbe Ergebnis liefert, wie die Standard CLEAN-Funktion `||`. Hierzu benutzen wir eine `nand` Funktion.

```
or :: Bool Bool -> Bool
or x y = nand (not x) (not y) where not x = nand x x
```

Nun können wir die Eigenschaft formulieren, die wir testen möchten, nämlich dass unsere Funktion für gleiche Eingabe das gleiche Ergebnis liefert wie der Operator `||` in CLEAN. Per Konvention setzt man als Präfix `prop` vor die Funktion die man testen möchte. In CLEAN würde das wie folgt aussehen.

```
propOr :: Bool Bool -> Bool
propOr x y = x||y == or x y
```

Auch für eigene Strukturen, z.B. einen Stack, kann man einfache zu testende Eigenschaften fordern.

```
:: Stack a :== [a]

pop :: (Stack a) -> Stack a
pop [_:r] = r

top :: (Stack a) -> a
top [a:_] = a

push :: a (Stack a) -> Stack a
push a s = [a:s]
```

Als zu testende Eigenschaft bietet sich an, die Funktionsweise der Stackoperationen zu prüfen. Bei polymorphen Typen, wie im Beispiel der Liste, muss man jedoch vorher einen konkreten Typ angeben, mit dem der Test durchgeführt werden soll. Wenn sich der Testlauf für einen konkreten Typ als erfolgreich erweist, so wird er für jeden Typ erfolgreich sein. Unten steht dabei `===` als generisches 'gleich'.

```
propStack :: a (Stack a) -> Bool | gEq{|*|} a
propStack e s = top (push e s) === e && pop (push e s) === s

propStackInt :: Int (Stack Int) -> Bool
propStackInt = propStack
```

Sind die Eigenschaften formuliert, kann man den Test starten. Dazu ruft man in CLEAN mit der `Start` Funktion den Testlauf auf.

```
Start = test propListInt
```

Wir haben hier recht einfache Tests durchgeführt. Zuerst wurde eine eigene Funktion mit einer eingebauten verglichen. Der zweite Test verifizierte die Richtigkeit einer Funktion für einen Datentyp. Ebenso kann man den Zusammenhang zwischen Ein- und Ausgabewerten oder ganze Module und Programme testen.

3.2 Generierung von Testdaten

Kommen wir nun zum Schritt zwei im gesamten Testprozeß, oder zum ersten Schritt, in den GAST eingreift, der Erstellung von Testdaten.

Die Testdaten werden hierbei nach einem bestimmten Verfahren bestimmt, bei dem Duplikate vermieden werden. Das erneute Testen mit demselben Datum ändert nämlich - und gerade bei referentiell transparanten Sprachen wie CLEAN - nichts am Ergebnis, ist somit nur zeitaufwändig und unerwünscht. Lediglich für Basistypen wie `Real` und `Int` werden Duplikate zugelassen. Den Autoren nach wäre es aufwendiger, eventuelle Duplikate zu eleminieren, als den Test doppelt auszuführen.

Wir müssen uns bewusst machen, dass es grundsätzlich zwei unterschiedliche Arten von Datentypen gibt. Endliche und unendliche. Vorsichtig muss man hier z.B. `Int` betrachten, von denen es je nach Architektur auf einem Rechner auch nur endlich viele Elemente gibt, z.B. 2^{32} oder 2^{64}, aber wie bei rekursiven Typen zu viele existieren, um wirklich alle zu testen.

Für einfache finite Typen wie `Bool` können wir also alle zu testenden Elemente erstellen, und somit sogar die Eigenschaft beweisen. Für rekursive und Typen mit vielen möglichen Elementen kann nur ein Teil der Menge getestet werden, und somit auch kein Beweis geführt werden, jedoch mit guter Wahrscheinlichkeit das richtige Ergebnis vorausgesagt werden. Dabei hat es sich erwiesen, dass es Sinn macht, zuerst bestimmte Grenzwerte zu generieren, und dann zufällige Werte, diese aber nicht doppelt. Viele Funktionen, gerade rekursive, sind nämlich meist so programmiert, dass sie Abbruchkriterien bei Grenzwerten der verwendeten Datentypen haben. Für Integer wären das also 0 oder 1, für Listen die leere Liste.

Die Autoren haben einen Algorithmus entwickelt, der genau dieses leistet. Dabei kann dieser Algorithmus für alle - auch selbst definierte - Datentypen Testdaten erstellen, dank der im vorigen Kapitel vorgestellten generischen Erweiterungen von CLEAN.

Dazu generiert GAST eine Liste der generischen Repräsentation der Werte eines Typs. Durch das generische System kann diese Repräsentation in 'echte' Werte umgeformt werden, mit denen dann Eigenschaften getestet werden.

Wie oben schon erwähnt, ist dieses für finite, einfache Typen eine leichte Aufgabe. Für `Color` würden wir eine Liste enthalten, die aus `Red`, `Yellow` und `Blue` besteht. Das läßt sich einfach aus der generischen Definition von

```
:: Colour° = EITHER UNIT (EITHER UNIT UNIT)|
```

und von den Farben

```
Red°    = LEFT UNIT
Yellow° = RIGHT (LEFT UNIT)
Blue°   = RIGHT (RIGHT UNIT)
```

herleiten.

Komplizierter wird es, wenn wir rekursive Typen betrachten. Solche kann man sich, wie in Kapitel 2 bei der Liste von Farben skizziert, als Baum vorstellen. Betrachten wir eine etwas andere Datenstruktur, die einen Baum darstellt.

```
Tree a = Tip | Node a (Tree a) (Tree a)
```

Würden wir einen generischen Baum dieser Datenstruktur erstellen, dann würde eine Listen mit Elementen, gingen wir nach Tiefensuche vor, wie folgt aussehen.

```
Tip, Node C Tip Tip, Node C (Node C Tip Tip) Tip,
Node C (Node C (Node C Tip Tip) Tip) Tip,
```

Da diese Datenstruktur kopfrekursiv ist, würde mit einem einfachen Algorithmus nie etwas wie `Node C Tip (Node C Tip Tip)` erreicht werden. Weiterer Nachteil ist, dass

'kleine' oder 'kurze' Werte, die meißt nahe links und rechts der Wurzel stehen, auch erst spät oder nie erreicht werden.

Eine andere Möglichkeit durch Bäume zu wandern, Breitensuche, führt leider auch nicht zur Lösung. Es ist nicht einfach, Bäume mit wachsender Tiefe auf diese Art zu generieren, ein großer Verwaltungsaufwand wäre vonnöten. Und nicht jeder so erstellte Baum wäre valid im Sinne der Definition der Datenstruktur.

Die Autoren wählten daher einen Algorithmus, der jedesmal zufällig einen Zweig auswählt, wenn er an ein `EITHER` in dem Baumtypen kommt. Um Duplikate auszuschließen, werden schon erstellte Bäume in einer Struktur `Trace` gespeichert.

```
:: Trace = Unit
         | Pair [(Trace,Trace)] [(Trace, Trace)]
         | Either Bool Trace Trace
         | Int [Int]
         | Done
         | Empty
```

Wir betrachten zunächst den Algorithmus `genAll`, der Einstiegspunkt für die Generierung der Testdaten - egal welchen Typs - ist.

```
1 genAll :: RandomStream -> [a] | generate{*} a
2 genAll rnd = g Empty rnd
3 where g Done rnd = []
4       g t    rnd = let (x, t2, rnd2) = generate{|*|} t rnd
5                        (t3, rnd3)    = nextTrace t2 rnd2
6                    in [x: g t3 rnd3]
```

Diese Funktion liefert eine Liste mit Elementen des gewünschten Typs, die für die Testdurchführung benötigt werden. Mit `generate{|*|}` wird ein konkretes Element erzeugt und mit `nextTrace` das nächste vorbereitet. Für selbst definierte Datentypen wird hier sichergestellt, dass keine doppelten erzeugt werden. Für `Int` wird hier im Prinzip nur das nächste Element einer zufälligen Liste geliefert.
Zum besseren Verständnis an dieser Stelle betrachten wir die Generierung von Integern. Dazu ist die Funktion `generate{|Int|}` wie folgt definiert:

```
1 generate{|Int|} t rnd = gen t rnd
2 where gen Empty       rnd = gen (Int predInts) rnd
3       gen (Int [])    rnd = let (i,rnd2) = genElem rnd in (i,t,rnd2)
4       gen (Int [i:r]) rnd = (i,Int [i:r],rnd)
```

Steigen wir in die Funktion `genAll` ein, so wird der erste Aufruf mit `Empty` als erstem Argument erfolgen und es wird Zeile 4 gematched. Der Aufruf dort von `generate{|*|}` führt in `generate{|Int|}` zum Ausführen von Zeile 2. `predInts` ist definiert als `[0,1,-1]`, also als typische Werte von Abbruchkriterien bei rekursiven Funktionen über Integern, wie schon oben erwähnt. So werden zunächst die Werte `0`, `1` und `-1` von `generate{|Int|}` zurückgegeben. Erst wenn diese vordefinierte Liste leer ist, Zeile 3 von `generate{|Int|}`, werden Zufallszahlen erzeugt. Hierbei kann es dann zu

Duplikaten kommen. Die Funktion `nextTrace` für Integer in `genAll` sieht wie folgt aus:

```
nextTrace (Int [])    rnd = (Int [], rnd)
nextTrace (Int [h:t]) rnd = (Int t, rnd)
```

Sie liefert also in Zeile 5 von `genAll` die Restliste ohne Kopf, mit der der nächste Durchlauf in `genAll` startet.

Wichtig ist hierbei, dass wir uns erneut klarmachen, dass die Auswertung - und damit die Generierung der Elemente - in CLEAN 'lazy' erfolgt, also nur soviel berechnet wird, wie zum Erstellen des nächsten Elements gebraucht wird. Benötigt die später vorgestellet Auswertungsfunktion keine neuen Elemente mehr, weil z.B. ein Gegenbeispiel gefunden ist, so werden auch keine Elemente mehr erzeugt.

Der Fall, dass Integer erzeugt werden ist relativ einfach, auch wird hier keine Vermeidung von Duplikaten vorgenommen, jedoch läßt sich an ihm anschaulich die Arbeitsweise erkennen. Komplizierter wird es, wenn benutzerdefinierte Datenstrukturen erstellt werden sollen. Die Funktion `nextTrace` muss nun für jeden ihrer Typen definiert sein.
Der interessante Fall ist `EITHER`, da hier im Baum verzweigt wird und das der Kern des Ansatzes ist, nämlich einerseits früh alle Werte nahe der Wurzel zu erreichen, aber andererseits den Verwaltungsaufwand gering zu halten und Duplikate auszuschließen. Für `EITHER` ist `nextTrace` folgend definiert:

```
 1 nextTrace (Either _ tl tr) rnd
 2   = let (b, rnd2) = genElem rnd in
 3     if b  (let (tl', rnd3) = nextTrace tl rnd2 in
 4             case tl' of
 5               Done = let (tr', rnd4) = nextTrace tr rnd3 in
 6                      case tr' of
 7                        Done = (Done, rnd4)
 8                        _    = (Either Right tl tr', rnd4)
 9                    = (Either Left tl' tr, rnd3))
10           (let (tr', rnd3) = nextTrace tr rnd2 in
11             case tr' of
12               Done = let (tl', rnd4) = nextTrace tl rnd3 in
13                      case tl' of
14                        Done = (Done, rnd4)
15                        _    = (Either Left tl' tr, rnd4)
16                    = (Either Right tl tr', rnd3))
```

Das Wichtigste findet hierbei in Zeile 2 statt. Mit dem generischen `genElem`, von dem in Zeile 3 ein `Bool` erwartet wird, wird entschieden, wie die Baumstruktur aufgebaut wird. Um den Algorithmus besser zu verstehen, betrachten wir noch die Definitionen von `nextTrace` für `Unit`:

```
nextTrace Unit rnd = (Done, rnd)
```

und genElem

```
class genElem a where genElem :: RandomStream -> .(a,RandomStream)

instance genElem Int  where genElem [r:rnd] = (r rem IntSize,rnd)
instance genElem Bool where genElem [r:rnd] = (isOdd r,rnd)
```

sowie die generate-Funktion für EITHER:

```
 1 generate{|EITHER|} fl fr Empty rnd
 2  = let (f,rnd2) = genElem rnd in
 3    if f (let (l,tl,rnd3) = fl Empty rnd2
 4          in (LEFT  l, Either Left  tl Empty, rnd3))
 5         (let (r,tr,rnd3) = fr Empty rnd2
 6          in (RIGHT r, Either Right Empty tr, rnd3))

 7 generate{|EITHER|} fl fr (Either left tl tr) rnd
 8  | left = let (l,tl2,rnd2) = fl tl rnd
 9           in (LEFT  l, Either left tl2 tr, rnd2)
10         = let (r,tr2,rnd2) = fr tr rnd
11           in (RIGHT r, Either left tl tr2, rnd2)
```

Wir veranschaulichen uns einen Durchlauf für das Beispiel

```
Color° = EITHER UNIT (EITHER UNIT UNIT)
```

Wieder starten wir bei genAll und rufen in Zeile 4 generate{|EITHER|} auf, welches uns, je nach dem Wert von f aus Zeile 2 in Zeile 4 oder 6 führt. Die Funktionen fl und fr sind die generate-Funktionen der Komponenten von EITHER, generieren also UNIT, ganuer gesagt generate{|UNIT|} _ rnd = (UNIT, Unit, rnd).
Ist f nun True, so liefert Zeile 3 (LEFT UNIT, Either Left Unit Empty, rnd). Das LEFT UNIT wird das x aus genAll und somit unser erstes zu testendes Element, es entspricht Red. Mit Either Left Unit Empty wird nextTrace aufgerufen. Auch hier wird zufällig entschieden, ob in Zeile 3 oder 9 fortgefahren wird. Sei es oBdA. Zeile 3. Es wird zunächst wieder nextTrace aufgerufen, diesmal jedoch mit Unit als Argument, was (Done, rnd) zurückliefert. Wir sind nun in Zeile 5, nextTrace wird mit Empty aufgerufen, was (Empty, rnd) als Rückgabe hat. Dadurch erhalten wir in Zeile 8 den Rückgabewert für den nextTrace-Aufruf in genAll, nämlich
(Either Right Done Empty, rnd).

Diese sagt aus, dass der linke Ast des generischen Baumes schon abgearbeitet ist, der Eintrag ist Done. Beim rechten Ast steht noch Empty, hier geht es im nächsten Aufruf von nextTrace in genAll weiter. Die generate{|EITHER|}-Funktion in genAll in diesem nächsten Aufruf erstellt dann, da Zeile 10 gematched wird - das Argument ist die Rückgabe aus dem letzten Aufruf, nämlich (Either Right Done Empty, rnd) - einen rechten Ast mit (RIGHT UNIT, Either right Done Unit).

So geht es weiter, bis alle Farben erstellt sind.

Selbstverständlich kann der Generator auch als Testdaten Funktionen erstellen. Dazu muss man allerdings explizit angeben, auf welchen Argumenten die Funktionen arbeiten sollen. Wenn wir eine `map`-Funktion testen wollten, müssten wie z.B. angeben, dass wir mit Funktionen auf Integern testen wollen. GAST würde nun nicht nur Integer zum Testen generieren, sondern auch Funktionen, die genau soviele Integer als Argumente erwarten, wie bei der Definition der `map`-Funktion angegeben ist. Die exakte Implementierung ist etwas komplizierter und es wird hier nicht weiter darauf eingegangen.

3.3 Durchführung der Tests

Die Durchführung der Tests ist nun relativ einfach zu bewerkstelligen. Wir haben die Eigenschaften formuliert, die getestet werden sollen, und die Möglichkeit, sinnvolle Testdaten zu erstellen.

Um die Testergebnisse zu speichern, wird eine Struktur erstellt, die festhält, wie das Ergebnis der einzelnen Tests war.

```
:: Admin = {res::Result, args::[String]}
:: Result = UnDef | OK | CE
```

Als nächstes wird eine Klasse von Funktionen `evaluate` für die Typen `Testable` definiert:

```
class Testable a where evaluate :: a RandomStream Admin -> [Admin]

instance Testable Bool
where evaluate b rs result
 = [{result & res = if b OK CE, args = reverse result.args}]

instance Testable Property
where evaluate (Prop p) rs result = p rs result

instance Testable (a->b) | Testable b & TestArg a
where evaluate f rs result
 = let (rs,rs2) = split rs in forAll f (genAll rs) rs2 result

forAll :: (a->b) [a] RandomStream Admin -> [Admin] | Testable b & TestArg a
forAll f list rs r = diagonal [apply f a (genRandInt seed) r \\ a<-list & seed<-rs ]

apply :: (a->b) a RandomStream Admin -> [Admin] | Testable b & TestArg a
apply f a rs r = evaluate (f a) rs {r & args = [show1 a:r.args]}
```

Grob gesagt leisten die Funktionen folgendes: Je nach Art der zu testenden Eigenschaft wird eine unterschiedliche Instanz von `evaluate` aufgerufen. Am Anfang wird in den meisten Fällen wohl eine Funktion zu testen sein, die dritte Instanz von `evaluate` wird

aufgerufen und die mit `genAll` erstellten Testdaten für die Eigenschaft angewendet. Die Funktionen `forAll`, `apply` und `diagonal` sorgen dabei dafür, dass auch Eigenschaften, die mehrere Argumente erwarten, versorgt werden. Dabei sorgt `diagonal` für ein systematisches Anwenden der Argumente (Cantorsche Diagonalfunktion). Mit `show1` wird das Argument als String gespeichert und für den nächsten Durchlauf mit übergeben, so dass man nachher auf Zwischenergebnisse zugreifen kann.

3.4 Ausgabe des Ergebnisses

Der letzte Schritt beim automatischen Testen ist - nach Erstellung der Testdaten und Durchführung der Tests - die Analyse des Ergebnisses. Dazu betrachten wir einfach die Nummer und Argumente des aktuellen Tests. Wenn Fehler auftreten, werden die Argumente, die diesen Fehler hervorriefen, ausgegeben. So kann besser nachvollzogen werden, woran der Test scheiterte. War der Test erfolgreich für eine Anzahl von Testdaten eines unendlichen oder rekursiven Datentyps, so kann ausgegeben werden, für wieviele Testläufe es erfolgreich war, für endliche Datentypen wird sogar Beweis ausgegeben.

Die Ausgabefunktion ist dabei wie folgt definiert:

```
 1 test :: RandomStream p -> [String] | Testable p
 2 test rs p = analyse (evaluate p rs newAdmin) maxTests maxArgs
 3 where analyse :: [Admin] Int Int -> [String]
 4       analyse [] n m = ["\nProof: success for all Arguments"]
 5       analyse l  0 m = ["\nPassed ", toString maxTests," tests"]
 6       analyse l  n 0 = ["\nPassed ", toString maxArgs," arguments"]
 7       analyse [res:rest] n m
 8         = [blank, toString (maxTests-n+1),":":showArgs res.args
 9           case res.res of
10             UnDef = analyse rest n (m-1)
11             OK    = analyse rest (n-1) (m-1)
12             CE    = ["\nCounterexample: ":showArgs res.args []]]
```

3.5 Zusätzliche Funktionen

Es gibt einige Möglichkeiten, die Tests zu verfeinern und die Ausgabe ausführlicher zu machen. Wenn man z.B. testet, ob ein Stack korrekt implementiert ist, kann man das unmöglich mit allen Testdaten durchführen. Dennoch, oder gerade deswegen, ist es oft hilfreich, Informationen über die getesteten Argumente zu erhalten. Einen Erweiterung leistet genau dieses, indem die oben vorgestellte `Admin`-Struktur erweitert wird. Eine mögliche Ausgabe für das Testen von `propStack` könnte folgendes sein.

```
Passed 4 tests
(0,[0,1]): 1 (25%)
(0,[0]): 1 (25%)
```

```
(0,[]): 1 (25%)
(1,[]): 1 (25%)
```

In diesem Fall war der Test erfolgreich, doch bei fehlgeschlagenen Tests wird so eine Fehlersuche deutlich vereinfacht. Eine genauere Definition der Testdaten kann z.B. bei rekursiven Funktionen nötig sein. Bei der naiven Definition der Fibonaccifunktion dauert es sehr lange, bis die Berechnung für große Werte ein Ergebnis liefert. Man kann deswegen die Menge der Argumente eingrenzen. Die Autoren stellen hierfür `For` zur Verfügung. Wenn die Eigenschaft für die Fibonaccifunktion mit `propFib` schon definiert ist, kann man mit

```
propFibR = propFib For [0..15]
```

den Bereich der Argumente auf 0 bis 15 beschränken.

In einem neueren Paper[4] stellen die Autoren einen neuen Algorithmus zum Generieren der Testdaten vor. Es wird von der Baumstruktur abgekehrt, stattdessen wird mit randomisierten Listen gearbeitet. Dieses geht deutlich schneller und einfacher.

Die folgende Liste gibt einen Ausschnitt einer möglichen Ausgabe für zufällig generierte Integer nach diesem Algorithmus. Auch hier werden die Grenzwerte zuerst generiert:

```
[0, -2147483648, 2147483647, -1, 1, 684985474,
 862966190, -1707763078, -930341561, 1734306050, ...]
```

4 Zusammenfassung

Mit GAST ist den Autoren ein interessantes und elegantes Testsystem gelungen, das die Stärken von funktionalen Sprachen voll ausschöpft und gleichzeitig angenehm übersichtlich bleibt.

Durch die Art, wie die Eigenschaften formuliert werden müssen, wird der Entwickler ein weiteres Mal gezwungen, sich mit seinem Code auseinander zu setzen. So werden eventuell schon erste Fehler aufgedeckt.

Die Stärke von GAST liegt an der Ausnutzung der generischen Fähigkeiten der Sprache, in der es geschrieben ist und in welcher auch die Programme selbst geschrieben sind, CLEAN. So werden Tests für praktisch jeden selbst definierten Datentyp möglich, alles vollautomatisch. Sollten einmal benutzerdefinierte Testdaten gewünscht werden, so ist dieses aber auch möglich.

Trotz der engen Verzahnung mit CLEAN kann GAST über Schnittstellen auch Code testen, der in einer fremden Sprache geschrieben ist.

GAST läßt ein effizientes Testen zu, da die Testdaten systematisch erstellt werden, und nicht nur zufällig, wie z.B. bei QUICKCHECK[5]. Es werden bekannte und bewährte Randwerte zuerst getestet und Duplikate vermieden.

GAST kann vier Arten von Fehlern erkennen, diese aber nicht automatisch unterscheiden, so dass bei der endgültigen Auswertung der Entwickler oder Tester gefragt ist. Zu den Fehlern gehören:

1. Fehler in der Implementierung - Diese sind es eigentlich, die man mit dem GAST-System finden möchte.

2. Fehler in der Spezifikation - Wird ein solcher Fehler festgestellt, muss man sorgfältiger bei der Spezifikation der Software arbeiten.

3. Fehler aufgrund der Genauigkeit der Maschine - Tritt z.B. beim Testen von Fließkommaberechnungen auf, man muss hier gewissen Toleranzen bei der Formulierung der Eigenschaften zulassen.

4. Nicht-terminierung bzw. Laufzeitfehler - Werden von GAST insofern aufgedeckt, da GAST die Argumente jedes Testlaufs zuerst ausgibt. So weiß man auch, welche Werte den Fehler hervorrufen.

Die Effizienz von GAST hängt hauptsächlich von der Auswertung der formulierten Eigenschaften ab, nicht von der Generierung der Daten. Normalerweise reichen 100 bis 1000 Tests, um eine wahre Aussage über eine Eigenschaft zu erhalten. Die Auswertung geht aufgrund der fehlenden Duplikate rasch von statten.

GAST ist kein Beweissystem, kann aber für finite Typen einen Beweis erbringen, indem es alle Eingaben testet. Dieses ist möglich, da es alle zu generierenden Testdaten erstellen kann und es sich auch merkt.

Literaturverzeichnis

[1] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In *14th International Workshop on Implementation of Functional Languages*, pages 84 – 100. Springer LNCS 2670, 2003.

[2] Rinus Plasmeijer and Markus van Eekelen. Clean version 2.0 language report. Technical report, 2001.

[3] Artem Alimarine and Marinus J. Plasmeijer. A generic programming extension for clean. In *Implementation of Functional Languages*, pages 168–185, 2001.

[4] Pieter Koopman and Rinus Plasmeijer. Generic generation of elements of types. In *6th Symposium on Trends in Functional Programming*, 2005.

[5] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.