Testing und Tracing

Diana Hille

14. Januar 2008

Motivation

Motivation

fehlerhaftes Programm:

```
main = let xs :: [Int]
           xs = [1,2,3]
           in print (head xs,last' xs)

last' (x:xs) = last' xs
last' [x] = x
```

Problem: Auffinden und Korrigieren des Fehlers in diesem Programm.

HAT allgemein

HAT ist ein Debugger

Ein Debugger (von engl. bug) ist ein Werkzeug zum Auffinden, Diagnostizieren und Beheben von Fehlern in Hardware und Software. http://de.wikipedia.org/wiki/Debugger

➥*Genauer:* HAT ist ein **Offline Tracer**

- Programm wird normal ausgeführt
- Zusätzlich zum Programmablauf wird Trace geschrieben
- Möglich Programme zu tracen, die
  - normal terminieren
  - mit Fehler terminieren
  - erst durch Benutzerabbruch terminieren

HAT Trail Allgemein

Beantwortung der Frage: „Wo kam das her?“

- Konsolenbefehl „hat-trail“
- Zeigt Werte, Ausdrücke, Ausgaben oder Fehlermeldungen
- Antwort: übergeordneter Aufruf oder Name

Erinnerung Testprogramm

```
main = let xs :: [Int]
           xs = [1,2,3]
           in print (head xs,last' xs)

last' (x:xs) = last' xs
last' [x] = x
```

Hat Trail Beispiel

Beispiel Ausgabe

```
Error: ------------------------------------------------
No match in pattern.
Output: ----------------------------------------------
(3,
Trail: ------------------ Last.hs line: 2 col: 12 ----
<- last' []
<- last' [_]
<- last' [_,_]
<- last' [3,_,_]
<- xs
```

HAT Observe Allgemein

Beantwortung der Frage „Was liefert mir der Aufruf einer bestimmen Funktion?“

- Konsolenbefehl „hat-observe“
- Beobachtung der übergeordneten Variablen (Funktionen, Konstanten)
- Alle Argumente und Resultate werden für eine Funktion gezeigt

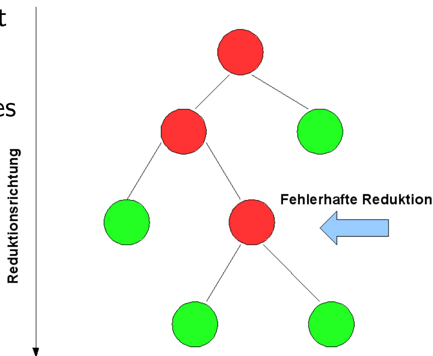
Einschränkung durch Muster der Syntax:

- identifier <pattern>* [=<pattern>][in <identifier>]

HAT Detect Allgemein

Wo kommt die falsche Ausgabe zustande?

- Konsolenbefehl: hat-detect
- Verwendung bei falscher Ausgabe eines Programmes
- Baumstruktur zum Auffinden von Fehlern
- Wurzelknoten: Reduktion von main



Beispiel

Standardsortierfunktion

Beispiel Ausgabe

```
$ hat-detect Example

      hat-detect 2.0x    (:h for help, :q to quit)

1   main = IO (print [3,3,3])    ? n
2   sort [3,2,1] = [3,3,3]    ? n
3   insert 1 [] = [1]    ? y
4   insert 2 [1] = [2,2]    ? n
5   insert 2 [] = [2]    ? y

Error located!
Bug found in reduction:    insert 2 [1] = [2,2]
```

Motivation

Liefert dieses Programm auf jede Eingabe die richtige Ausgabe?

Motivation

reverse (reverse xs) = xs

Lösung:

Automatisches Testen durch *Quickcheck*

- Werkzeug, das automatisch Testfälle generiert
- Testdatengenierungssprache bereits in Haskell eingebettet (*Lightweight*)

Achtung:

Ein postitives Testergebnis ist *keine* Garantie für Fehlerfreiheit

Gesetze

Gesetze am Beispiel

Beispiel Gesetze

```
import QuickCheck

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

main =
   do quickCheck prop_ReverseReverse
```

Positives Ergebnis:

Positives Ergebnis

```
Main> main
OK, passed 100 tests.
```

Überladung

Problem:

- QuickCheck ist überladen um allen Gesetzen gerecht zu werden

Testen von Assoziativität über Typfunktionen

Assoziativität

```
prop_Assoc :: Eq a => (a -> a) -> a -> a -> a -> Bool
prop_Assoc f x y z =
   f x (f y z) == f (f x y) z

prop_AssocPlusInt = prop_Assoc ((+) :: Int -> Int -> Int)
prop_AssocPlusDouble = prop_Assoc ((+) :: Double -> Double
->Double)
```

Problem: Assoziativität nur für Integer möglich

Bedingungsgesetze

Gesetze halten unter bestimmten Bedingungen

Quickchek bietet ein Folgerungskombinator für Bedingungsgesetze

Beispiel

Gesetz:

x <= y $\Longrightarrow$ max x y == y

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

Typ wurde von Bool zu Property geändert

➥ Testsemantik ist anders bei Bedingungsgesetzen

➥ Anstelle von 100 zufälligen Tests, werden 100 Test durchgeführt, die die Bedingung erfüllen

Testdaten

Erfassen von Testdaten:

Modifiziertes prop_Insert

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

Quickcheck liefert:

Ausgabe

OK, passed 100 tests (43% trivial).

Problem:
Viele Tests des Einfügens in die leere Liste

Mehr Informationen über prop_Insert

Histogramm von Werten:

Modifiziertes prop_Insert

Programm:

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
 ordered xs ==>
  collect (lenght xs) $
   ordered (insert x xs)
```

Mögliches Resultat:

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

Lösung des Problems von prop_Insert

Benutzen des üblichen Testdatengenerators orderedList:

Modifiziertes prop_Insert

Programm:

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

Mögliches Resultat:

OK, passed 100 tests.

Quickcheck bietet aber auch die Möglichkeit eigene Generatoren zu definieren

Definition von Generatoren

Arbitrary

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

newtype Gen a = Gen (Rand -> a)

choose :: (Int, Int) -> Gen Int

return :: a -> Gen a
(>>=) :: Gen a -> (a -> Gen b) -> Gen b

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

Definition von Generatoren

Anwendungsbeispiel:

Binäre Bäume

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

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

Problem: Diese Definition hat nur eine 50%ige Chance der Terminierung

Definition von Generatoren

Lösung des Problems bei den Binären Bäumen:

➥ Limitation der Größe der zu generierenden Testdaten

Definition eines neuen Kombinators

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

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

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))) ]
```

Zusammenfassung HAT

Wozu tracen?

- HAT ist ein Programm direkt für Haskell
- Auffinden von Fehler auch in größeren Programmen wird erleichtert
- Das Programm wird bereits einmal probeweise ausgeführt
- HAT bietet verschiedene Möglichkeiten der Fehlersuche

Zusammenfassung Quickcheck

Wozu testen?

- Testen 40-50% der Entwicklungskosten eines Softwareprojektes aus
- Zeigt deutlich, ob ein Fehler vorhanden ist
- Garantiert keine Fehlerfreiheit
- Quickcheck ist bereits in Haskell implementiert
- Es muss keine neue Programmiersprache gelernt werden
- Quickcheck lässt sich modifizieren

Ende

Vielen Dank für die Aufmerksamkeit!
Fragen?