Haskell Programming With Tests, and Some Alloy

Jan van Eijck

jve@cwi.nl

Master SE, 2010

Abstract

How to write a program in Haskell, and how to use the Haskell testing tools ...

QuickCheck is a tool written in the functional programming language Haskell that allows testing of specifications by means of randomly generated tests. QuickCheck is part of the standard Haskell library. Re-implementations of QuickCheck exist for many languages, including Ruby and Scheme.

SmallCheck is a similar tool, different from QuickCheck in that it tests properties for all finitely many values of a datatype up to some given depth, with progressive increase of depth.

Haskell is a research language: many of the testing tools that were first developed for Haskell later find their way to other languages.

These slides discuss QuickCheck (two versions), SmallCheck, and some work in progress.

We end with some examples of Alloy specifications.

How to Write a Haskell Program

http://www.haskell.org/haskellwiki/How_to_write_a_Haskell_program

Influence of QuickCheck

http://www.sigplan.org/award-icfp.htm

SIGPLAN Most Influential ICFP Paper Award 2010 (for 2000): Quickcheck: a lightweight tool for random testing of Haskell programs, Koen Claessen and John Hughes

"This paper presented a very simple but powerful system for testing Haskell programs that has had significant impact on the practice of debugging programs in Haskell. The paper describes a clever way to use type classes and monads to automatically generate random test data. QuickCheck has since become an extremely popular Haskell library that is widely used by programmers, and has been incorporated into many undergraduate courses in Haskell. The techniques described in the paper have spawned a significant body of follow-on work in test case generation. They have also been adapted to other languages, leading to their commercialisation for Erlang and C."

QuickCheck: Background

See the QuickCheck webpage at http://www.cs.chalmers.se/~rjmh/QuickCheck/.

From this page:

QuickCheck is a tool for testing Haskell programs automatically. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases.

Specifications are expressed in Haskell, using combinators defined in the QuickCheck library.

QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

QuickCheck has excellent documentation on the Haskell homepage, for

it is a standard Haskell library. See

http://haskell.org/haskellwiki/Introduction_to_QuickCheck

From this: "QuickCheck is effectively an embedded domain specific language for testing Haskell code."

More information on the QuickCheck website http://www.cs.chalmers.se/~rjmh/QuickCheck/ and in the papers Claessen and Hughes [2000] Claessen and Hughes [2003] (see link on website). An online manual is at http://www.cs.chalmers.se/~rjmh/QuickCheck/manual.html.

A Simple Example

A simple example of a property definition is

```
prop_RevRev xs = reverse (reverse xs) == xs
  where types = xs::[Int]
```

To check the property, we load this definition in to hugs and then invoke

```
Main> quickCheck prop_RevRev
OK, passed 100 tests.
```

When a property fails, QuickCheck displays a counter-example. For example, if we define

```
prop_RevId xs = reverse xs == xs
  where types = xs::[Int]
```

then checking it results in

```
Main> quickCheck prop_RevId
Falsifiable, after 1 tests:
[-3,15]
```

Chapter on Testing in Real World Haskell

http://book.realworldhaskell.org/read/testing-and-quality-as
html

Simplification of CounterExamples

QuickCheck 2 computes minimal counterexamples.

Method: simplify a counterexample and check if the test still fails.

A counterexample is minimal if it fails the test but all its simplifications succeed.

See Hughes [2007].

SmallCheck

http://www.cs.york.ac.uk/fp/smallcheck/

http://hackage.haskell.org/cgi-bin/hackage-scripts/package/smallcheck

Based on the 'small domain hypothesis' that is also the chief article of faith behind Alloy.

See Runciman et al. [2008]

Irulan (work in progress)

http://www.doc.ic.ac.uk/~tora/irulan/

Draft paper: http://www.doc.ic.ac.uk/~tora/irulan/draft.pdf

Contract Checking for Haskell

Xu et al.

And a bit of Alloy . . .

- Alloy for automated logical reasoning
- Alloy specifications of algorithms
- On your to do list:
  - Look through the example code in these slides,
  - make sure you understand what is happening.

Checking the Consistency and the Consequences of a Story

Consider the following story:

> John likes all girls. Some girl likes everybody but John. Mary likes only John. No boy except John likes every girl.

1. Use Alloy to check whether this is a consistent story.
2. Use Alloy to check whether it follows from the above that John is the only boy.

Crime Scene Investigation With Alloy

Story about witnesses, their assertions, and meta-information about how many of them are truthful.

From this the consistent scenarios can be generated with Alloy . . . How?

Reasoning about Repetition: Finding Celebrities

Let us assume that a celebrity (within a certain circle of people) is characterized by the following two properties:

- everybody knows the celebrity,
- the celebrity does not know anyone else.

(Whether the celebrity knows him- or herself does not matter.)

Here is the problem we are going to solve. In a company of about one thousand people there is a celebrity. The task is to spot this person by repeatedly asking the question 'do you know that person?'

Naive solution

Pick a candidate-celebrity X, and ask for each of the others whether X knows them. If X happens to know anyone, we go on with the next candidate. If it turns out that X knows noone, we have found our celebrity.

If you are out of luck the first candidate knows of the 999 others precisely the last one, and you need to pose 999 questions to rule her out. Similarly, for the second candidate it might take at worst 998 questions to rule him out, and so on. Thus, this worst-case scenario will take

$$999 + 998 + 997 + \cdots + 1 = (500 * 1001) - 1000 = 499500$$

questions (almost a half million).

Better Solution

Given any two people x and y there are two outcomes for the question whether x knows y:

- x knows y then x is not the celebrity.
- x does not know y then y is not the celebrity.

The following procedure uses this.

Algorithm for Finding Celebrities

Start with the list of all people in the domain. Since it is given that the domain contains a celebrity, the list is not empty.

- While the list has more than one member,
  - take the first two elements x, y from the list;
  - if x knows y then remove x from the list;
    otherwise, remove y from the list

The celebrity is the single person that remains on the list.

Finding the Celebrity: Alloy Specification

```
module myexamples/famous

open util/ordering[State] as so

sig Person { know: set Person }

one sig Famous in Person {}

fact lonelyAtTheTop {
  all x: Person - Famous | x !in Famous.know
  all x: Person - Famous | x in know.Famous
}
```

```
sig State { mark: set Person }

// at initialisation everybody is marked
pred init { let fs = so/first | fs.mark=Person }

pred extend [pre, post: State] {
  some x: pre.mark, y: pre.mark - x |
     y in x.know => post.mark = pre.mark - x
     else post.mark = pre.mark - y
}

fact createStates {
  init
  all s: State - so/last |
         let s' = so/next[s] | extend[s,s']
}
```

```
assert finalMarkFamous {
  let final = so/last | final.mark = Famous
}

run {} for 10

check finalMarkFamous for 5
```

Matchmaking

An internet dating program should match candidates looking for a date, in accordance with a list of criteria.

Make this informal specification more precise, and implement it in Ruby or Haskell.

Instead, we are going to work out the specification in Alloy. Here are the criteria.

- Compatible sexual preferences (match gay men with gay men, heterosexual men with heterosexual women (and vice versa), and lesbian women with lesbian women,
- Age difference in male/female matches no more than ten years if the male is older, no more than five years if the female is older. For cases of m/m and f/f matches: age differences no more than five years.
- Similar level of education (distinguish three levels).
- Similar preferences (or at least reasonable overlap in preferences), with preferences ranging over the following list: (i) Concerts and Museums, (ii) Going out, (iii) Sports, (iv) Conversation, (v) Travel.

```
module myexamples/matchmaking

open util/relation as rel
open util/ordering[Age] as order

abstract sig Person {
  age: Age,
  edu: Edu,
  i:   set Interest,
  match: set Person
}

abstract sig Male extends Person {}
abstract sig Female extends Person {}

sig Straight, Gay in Person {}
```

```
fact { Person - Straight = Gay }
fact { some Gay }

sig Age {}

abstract sig Edu {}
one sig Basic, Middle, Higher extends Edu {}

abstract sig Interest {}
one sig Concerts, GoingOut, Sports, Conversation, Travel
   extends Interest {}

fact matchConstraints {
   irreflexive[match]
   symmetric[match]
   all x,y: Person | y in match[x] => edu[x] = edu[y]
```

```
all x,y: Person | y in match[x]
                  and x in Male and y in Female
                  => x+y in Straight
all x,y: Male   | y in match[x] => x+y in Gay
all x,y: Female | y in match[x] => x+y in Gay
all x,y: Person | y in match[x] =>
                        age[x] = age[y]
                        or age[x].prev = age[y]
                        or age[y].prev = age[x]
                        or (x in Male
                              and y in Female
                              and
                              age[x].prev.prev = age[y])
all x,y: Person | y in match[x]
                  => i[x] = i[y] or #(i[x] & i[y]) > 2
```

```
}

run { total[match,Person]}
     for exactly 10 Person, exactly 5 Age
```

References

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs,. In Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000.

Koen Claessen and John Hughes. Specification based testing with QuickCheck. In The Fun of Programming, Cornerstones of Computing, pages 17–40. Palgrave, 2003.

John Hughes. Quickcheck testing for fun and profit. In Michael Hanus, editor, Practical Aspects of Declarative Languages; 9th International Symposium, PADL 2007, Nice, France, volume 4353 of Lecture Notes in Computer Science, pages 1–32. Springer, 2007.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values.

In Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell, pages 37–48, New York, NY, USA, 2008. ACM.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell.