QuickCheck - Random Property-based Testing

Why Functional Programming And In Particular QuickCheck Is Cool

Arnold Schwaighofer

Institute for Formal Models and Verification
Johannes Kepler University Linz

26 June 2007 / KV Debugging
Outline

Haskell - A Truly (Cool) Functional Programming Language
  What - Functional Programming
  How - Functional Programming
  Why - Functional Programming

Quickcheck - A Truly Cool Property Based Testing Tool
  What - Random Property Based Testing
  How - Random Property Based Testing
  Why - Random Property Based Testing

Success Stories

Related Work And Outlook

Summary
Outline

Haskell - A Truly (Cool) Functional Programming Language
  What - Functional Programming
  How - Functional Programming
  Why - Functional Programming

Quickcheck - A Truly Cool Property Based Testing Tool
  What - Random Property Based Testing
  How - Random Property Based Testing
  Why - Random Property Based Testing
  Success Stories
  Related Work And Outlook

Summary
Outline

Haskell - A Truly (Cool) Functional Programming Language
  What - Functional Programming
  How - Functional Programming
  Why - Functional Programming

Quickcheck - A Truly Cool Property Based Testing Tool
  What - Random Property Based Testing
  How - Random Property Based Testing
  Why - Random Property Based Testing
  Success Stories
  Related Work And Outlook

Summary
What Is Functional Programming?
From a lazy perspective

As in features of the programming language Haskell [JH99].

- Functions are first class - values that can be passed around
- Referential integrity - no side effects!
- Pattern matching - Write functions according to the type’s data constructor
- Laziness - evaluate terms when they are needed (and only once)
- Statically typed - all terms must have a valid type at compile time
Functions As Essential Building Blocks

Functions are curried

\[ \text{multiply} \quad :: \quad \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \]
\[ \text{multiply} \ x \ y = x \times y \]

Functions build of other functions

\[ \text{multiplyByTwo} \quad :: \quad \text{Integer} \rightarrow \text{Integer} \]
\[ \text{multiplyByTwo} \ x = \text{multiply} \ 2 \ x \]

Functions can be polymorphic

\[ \text{id} \quad :: \quad a \rightarrow a \]
\[ \text{id} \ x = x \]

\[ \text{applyTwice} \quad :: \quad (a \rightarrow a) \rightarrow a \rightarrow a \]
\[ \text{applyTwice} \ f \ x = f \ (f \ x) \]

file:///Users/arnold/Desktop/qc-ex/01-func.hs
A Repeating Pattern

Functions are defined using pattern matching on the argument type(s).

```
data List a = Empty | Prepend a (List a)
List a ≡ [a]
Empty ≡ []
Prepend x xs ≡ x : xs
```

```
[1, 2, 3, 4] ≡ 1 : (2 : (3 : (4 : [])))
```

```
len :: [a] → Integer
len [] = 0
len (x : xs) = 1 + len xs
```

file:///Users/arnold/Desktop/qc-ex/02-pattern.hs
The Beauty Of Being Lazy
Functions are lazy. Only evaluate when result is needed.

\[
\text{allNumbersFrom} :: \text{Integer} \rightarrow [\text{Integer}]
\]
\[
\text{allNumbersFrom} \ x = x : \text{allNumbersFrom} \ (x + 1)
\]

\[
\text{first} :: [\text{Integer}] \rightarrow \text{Integer}
\]
\[
\text{first} \ [ ] = [ ]
\]
\[
\text{first} \ (x : xs) = x
\]

\[
\text{take} :: \text{Integer} \rightarrow [a] \rightarrow [a]
\]
\[
\text{take} \ 0 \ _ = [ ]
\]
\[
\text{take} \ _ \ [ ] = [ ]
\]
\[
\text{take} \ n \ (x : xs) = x : (\text{take} \ (n - 1) \ xs)
\]

\[
\text{first} \ (\text{allNumbersFrom} \ 1) \equiv 1
\]
\[
\text{take} \ 5 \ (\text{allNumbersFrom} \ 1) \equiv [1, 2, 3, 4, 5]
\]

file:///Users/arnold/Desktop/qc-ex/03-lazy.hs
Advantages Of Functional Programming

- Modular programming - higher order functions, producer consumer pattern due to laziness [Hug89]
- Conciseness - less to write, less to read
- Easier to debug - because functions are pure
- Easier to test - because functions are pure
- Safer - Type checking finds a lot of errors before even running the program [Car97]
- Typed Lambda Calculus [Chu36]- mathematical theory more beautiful than Turing Machine (to me at least)
QuickCheck [CH00] What Is It About?

- Traditionally test cases are written by hand (Unit testing)
- Tedious work - remember we are in a lazy setting
- Idea: Functional setting, dependency only on arguments
- Specify properties of a function (e.g. $f(x) > 0$ for all $x$)
- Randomly generate arguments and check that property holds
- We can do this even for arguments that are functions (remember Haskell is cool)
Property Based Testing

Test case generator

Function under test

Property

OK?

[Cla06]
Simple Tests

Random test generators for most built-in types (Integers, Boolean, Tuples, Lists) are predefined.

\[
\begin{align*}
\text{prop}_\text{RevUnit} & \quad :: \text{Integer} \rightarrow \text{Bool} \\
\text{prop}_\text{RevUnit} x = & \quad \text{reverse} \ [x] \equiv [x] \\
\text{prop}_\text{RevRev} & \quad :: \text{[Integer]} \rightarrow \text{Bool} \\
\text{prop}_\text{RevRev} xs = & \quad \text{reverse} \ (\text{reverse} \ xs) \equiv xs \\
\text{prop}_\text{RevApp} & \quad :: \text{[Integer]} \rightarrow \text{[Integer]} \rightarrow \text{Bool} \\
\text{prop}_\text{RevApp} xs ys = & \quad \text{reverse} \ (xs \uplus ys) \equiv \text{reverse} \ ys \uplus \text{reverse} \ xs
\end{align*}
\]

file:///Users/arnold/Desktop/qc-ex/04-qcsimple.hs
We Don’t Stop At Functions ’Cause Remember We Are Cool

Extensionality on functions.

\[(f \equiv g) \ x = f \ x \equiv g \ x\]

To show function composition is associative.

\[prop\_CompositionAssociative :: (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Bool}\]

\[prop\_CompositionAssociative \ f \ g \ h = (f \circ (g \circ h)) \equiv (f \circ g) \circ h\]

file://Users/arnold/Desktop/qc-ex/05-qcfunc.hs
Observing What Is Going On

There are combinators that can be used in specifications that tell us what is going on.

\[
\text{prop\_Insert} :: \text{Int} \to [\text{Int}] \to \text{Property} \\
\text{prop\_Insert } x \; xs = \\
\text{ordered } xs \implies \\
\text{collect } (\text{length } xs) \backslash$
\[
\text{ordered } (\text{insert } x \; xs)
\]

OK, passed 100 tests.
20% 0.
10% 1.
9% 3.
...
1% 16.
Generating Random Data

QuickCheck provides support for user defined random data generators.

- User defined types (structures)
- Control the size of the generated data
- Control the distribution of generated data
Do We Really Want To test This Way?

- Yes, because less work then writing unit tests.
- Find errors in functions, also in corner cases which unit test might have forgotten
- Properties serve as documentation
- Find errors in specification
- Don’t need to learn another language for specification, expressed in Haskell
Is It Really Used In Practice?

- Ships with all major Haskell compilers (Hugs, GHC, NHC)
- Used in many Haskell libraries and applications (e.g. Edison - a functional data structures library, xmonad - a functional window manager)
- Commercial version for Erlang (concurrent functional language) - called Quviq QuickCheck
- Quviq QuickCheck will be use in new product development at Erricson (Telecommunication products) [AHJW06]
- Versions for Erlang, Scheme, Python, ML, Lisp, ocaml
What Is Everybody Else Doing?

- HUnit - a unit testing framework based on JUnit, no automatic generation of test cases [Her02]
- JML - Java Modelling Language [LBR99] allows specification, verification using tools like KeY [BHS07], ESC/Java2 [CK04]
- Extend Static Checking for Haskell, implementation of Pre/Postcondition reasoning (Hoare calculus) for Haskell verified using symbolic evaluation [Xu06]

The authors of QuickCheck are looking into ways to integrate QuickCheck with Hat. Hat is a tracing tool. When a test fails the tracer would be entered and the programmer could look at the computation. [CH02]
What to remember?

- Functional programs are easier to test/debug - no global state
- Functional programs are concise and modular
- Functional programming is cool. If only to learn new kinds of abstractions (Sapir-Whorf hypothesis)
- Property based random testing is good to test functions with minimal effort
- But also serves as documentation
Thank you!
Defining Your Own data Generator

Must implement instance of type class Arbitrary.

class Arbitrary a where
    arbitrary :: Gen a

Using e.g oneof.

data Color = Red | Green | Blue

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

Or controlling the frequency of choice.

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

instance Arbitrary a ⇒ Arbitrary [a] where
    arbitrary = frequency
        [(1, liftM Leaf arbitrary),
         (2, liftM2 Branch arbitrary arbitrary)]
I Wan To learn More About Haskell And Functional Programming!

- Or find other resources on [http://www.Haskell.org](http://www.Haskell.org)
- More info concerning type systems in Types and Programming Languages [Pie02]
- An eye opener: Structur and Interpretation of Computer Programs [ASS96]


Koen Claessen and John Hughes.
Quickcheck: a lightweight tool for random testing of haskell programs.


K. Claessen and J. Hughes.
Testing monadic code with quickcheck.

Alonzo Church.
An unsolvable problem of elementary number theory.

David R. Cok and Joseph R. Kiniry.

Koen Claessen.
Quickcheck: Property-based random testing.
*Advanced Functional Programming*, Chalmers University of Technology, Gothenburg, Sweden, August 2006.

Dean Herington.
Hunit 1.0, 2002.
Retrieved on 25 June 2007, URL: 
http://hunit.sourceforge.net/HUnit-1.0/Guide.html.
John Hughes.
Why Functional Programming Matters.

Simon Peyton Jones and John Hughes.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby.
JML: A notation for detailed design.

Benjamin Pierce.
*Types and Programming Languages*, chapter 1.1, page 1.

Dana N. Xu.
Extended static checking for haskell.