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

Outline

Outline

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

$$\begin{aligned} \mathit{multiply} &:: \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \\ \mathit{multiply}\ x\ y &= x * y \end{aligned}$$

Functions build of other functions

$$\begin{aligned} \mathit{multiplyByTwo} &:: \mathit{Integer} \rightarrow \mathit{Integer} \\ \mathit{multiplyByTwo}\ x &= \mathit{multiply}\ 2\ x \end{aligned}$$

Functions can be polymorphic

$$\begin{aligned} \mathit{id} &:: a \rightarrow a \\ \mathit{id}\ x &= x \end{aligned}$$

$$\begin{aligned} \mathit{applyTwice} &:: (a \rightarrow a) \rightarrow a \rightarrow a \\ \mathit{applyTwice}\ f\ x &= f\ (f\ x) \end{aligned}$$

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

A Repeating Pattern

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

$$
\begin{aligned}
\textbf{data}\ \mathit{List}\ a &= \mathit{Empty} \mid \mathit{Prepend}\ a\ (\mathit{List}\ a) \\
\mathit{List}\ a &\equiv [a] \\
\mathit{Empty} &\equiv [\,] \\
\mathit{Prepend}\ x\ xs &\equiv x : xs
\end{aligned}
$$

$$[1,2,3,4] \equiv 1 : (2 : (3 : (4 : [\,])))$$

$$
\begin{aligned}
\mathit{len} &:: [a] \rightarrow \mathit{Integer} \\
\mathit{len}\ [\,] &= 0 \\
\mathit{len}\ (x : xs) &= 1 + \mathit{len}\ xs
\end{aligned}
$$

file:///Users/arnold/Desktop/qc-ex/02-pattern.hs

The Beauty Of Being Lazy

Functions are lazy. Only evaluate when result is needed.

$$
\begin{aligned}
&\mathit{allNumbersFrom} :: \mathit{Integer} \to [\mathit{Integer}] \\
&\mathit{allNumbersFrom}\ x = x : \mathit{allNumbersFrom}\ (x + 1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{first} :: [\mathit{Integer}] \to \mathit{Integer} \\
&\mathit{first}\ [\,] = [\,] \\
&\mathit{first}\ (x : xs) = x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{take} :: \mathit{Integer} \to [a] \to [a] \\
&\mathit{take}\ 0\ \_ = [\,] \\
&\mathit{take}\ \_\ [\,] = [\,] \\
&\mathit{take}\ n\ (x : xs) = x : (\mathit{take}\ (n - 1)\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{first}\ (\mathit{allNumbersFrom}\ 1) \equiv 1 \\
&\mathit{take}\ 5\ (\mathit{allNumbersFrom}\ 1) \equiv [1, 2, 3, 4, 5]
\end{aligned}
$$

file:///Users/arnold/Desktop/qc-ex/03-lazy.hs

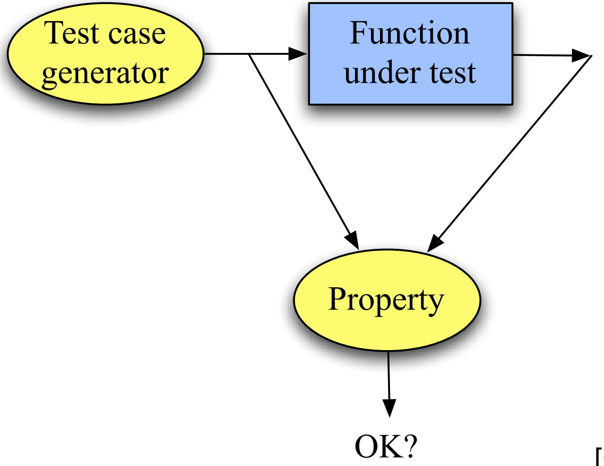
Advantages Of Functional Programming

- 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



[Cla06]

Simple Tests

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

$$\begin{aligned}&\mathit{prop\_RevUnit} :: \mathit{Integer} \rightarrow \mathit{Bool}\\&\mathit{prop\_RevUnit}\ x =\\&\quad \mathit{reverse}\ [x] \equiv [x]\end{aligned}$$

$$\begin{aligned}&\mathit{prop\_RevRev} :: [\mathit{Integer}] \rightarrow \mathit{Bool}\\&\mathit{prop\_RevRev}\ \mathit{xs} =\\&\quad \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \equiv \mathit{xs}\end{aligned}$$

$$\begin{aligned}&\mathit{prop\_RevApp} :: [\mathit{Integer}] \rightarrow [\mathit{Integer}] \rightarrow \mathit{Bool}\\&\mathit{prop\_RevApp}\ \mathit{xs}\ \mathit{ys} =\\&\quad \mathit{reverse}\ (\mathit{xs} +\!\!+\ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{ys} +\!\!+\ \mathit{reverse}\ \mathit{xs}\end{aligned}$$

`file:///Users/arnold/Desktop/qc-ex/04-qcsimple.hs`

We Don't Stop At Functions 'Cause Remember We Are Cool

Extensionality on functions.

$$(f === g)\ x = f\ x \equiv g\ x$$

To show function composition is associative.

$$\begin{aligned} prop\_CompositionAssociative :: &(Int \rightarrow Int) \rightarrow \\ &(Int \rightarrow Int) \rightarrow \\ &(Int \rightarrow Int) \rightarrow \\ &Int \rightarrow Bool \\ prop\_CompositionAssociative\ f\ g\ h = & \\ \quad f \circ (g \circ h) === (f \circ g) \circ h & \end{aligned}$$

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.

$$
\begin{aligned}
&\mathit{prop\_Insert} :: \mathit{Int} \rightarrow [\mathit{Int}] \rightarrow \mathit{Property} \\
&\mathit{prop\_Insert}\ x\ xs = \\
&\quad \mathit{ordered}\ xs ==> \\
&\qquad \mathit{collect}\ (\mathit{length}\ xs)\ \backslash\$ \\
&\qquad\quad \mathit{ordered}\ (\mathit{insert}\ x\ xs)
\end{aligned}
$$

```
OK, passed 100 tests.
20% 0.
10% 1.
9% 3.
...
1% 16.
```

file:///Users/arnold/Desktop/qc-ex/06-monitor.hs

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.

$$\textbf{class } Arbitrary\ a \textbf{ where}$$
$$\quad arbitrary :: Gen\ a$$

Using e.g oneof.

$$\textbf{data } Color = Red \mid Green \mid Blue$$
$$\textbf{instance } Arbitrary\ Color \textbf{ where}$$
$$\quad arbitrary = oneof$$
$$\qquad [return\ Red, return\ Green, return\ Blue]$$

Or controlling the frequency of choice.

$$\textbf{data } Tree\ a = Leaf\ a \mid Branch\ (Tree\ a)\ (Tree\ a)$$
$$\textbf{instance } Arbitrary\ a \Rightarrow Arbitrary\ [a] \textbf{ where}$$
$$\quad arbitrary = frequency$$
$$\quad [(1, liftM\ Leaf\ arbitrary),$$
$$\qquad (2, liftM2\ Branch\ arbitrary\ arbitrary)]$$

I Wan To learn More About Haskell And Functional Programming!

- For the lazy `http://video.s-inf.de/`. Look for “Grundlagen der Funktionalen Programmierung”.
- Or find other resources on `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]

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger.
Testing telecoms software with quviq quickcheck.
In Phil Trinder, editor, *Proceedings of the Fifth ACM SIGPLAN Erlang Workshop*. ACM Press, 2006.

Harold Abelson, Gerald Jay Sussman, and Julie Sussman.
Structure and Interpretation of Computer Programs.
McGraw Hill, Cambridge, Mass., second edition, 1996.
Available online: http://mitpress.mit.edu/sicp/full-text/book/book.html.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors.
Verification of Object-Oriented Software: The KeY Approach.
LNCS 4334. Springer-Verlag, 2007.

Luca Cardelli.
Type systems.
In Allen B. Tucker, editor, *The Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.
URL: http://citeseer.ist.psu.edu/cardelli97type.html.

Koen Claessen and John Hughes.

Quickcheck: a lightweight tool for random testing of haskell programs.
ACM SIGPLAN Notices, 35(9):268–279, 2000.

K. Claessen and J. Hughes.
Testing monadic code with quickcheck.
In *Proceedings of the ACM SIGPLAN 2002.*, 2002.

Alonzo Church.
An unsolvable problem of elementary number theory.
American Journal of Mathematics, 58:345–363, 1936.

David R. Cok and Joseph R. Kiniry.
Esc/java2: Uniting esc/java and jml - progress and issues in building and using esc/java2, 2004.

Koen Claessen.
Quickcheck: Property-based random testing.
Advanced Functional Programming, Chalmers University of Technology, Goethenburg, 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.
Computer Journal, 32(2):98–107, 1989.

Simon Peyton Jones and John Huges.
Haskell 98 report, February 1999.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby.
JML: A notation for detailed design.
In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

Benjamin Pierce.
Types and Programming Languages, chapter 1.1, page 1.
The MIT Press, 2002.
URL: http://www.cis.upenn.edu/ bcpierce/tapl/index.html.

Dana N. Xu.
Extended static checking for haskell.
In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59, New York, NY, USA, 2006. ACM Press.