COMP4075: Lecture 14
Property-based Testing

Henrik Nilsson

University of Nottingham, UK

QuickCheck: What is it? (1)

- Framework for property-based testing
- Flexible language for stating properties
- Random test cases generated automatically based on type of argument(s) to properties.
- Highly configurable:
  - Number, size of test cases can easily be specified
  - Additional types for more fine-grained control of test case generation
  - Customised test case generators

QuickCheck: What is it? (2)

- Support for checking test coverage
- Counterexample produced when test case fails
- Counterexamples automatically shrunk in attempt to find minimal counterexample

Basic Example

$$\textbf{import}\ Test.QuickCheck$$

$$prop\_RevRev :: [Int] \rightarrow Bool$$

$$prop\_RevRev\ xs =$$

$$\quad reverse\ (reverse\ xs) \equiv xs$$

$$prop\_RevApp :: [Int] \rightarrow [Int] \rightarrow Bool$$

$$prop\_RevApp\ xs\ ys =$$

$$\quad reverse\ (xs +\!\!+\ ys) \equiv reverse\ ys +\!\!+\ reverse\ xs$$

$$quickCheck\ (prop\_RevRev\ .\&\&.\ prop\_RevApp)$$

Basic Example

$$\textbf{import}\ \mathit{Test.QuickCheck}$$

$$\mathit{prop\_RevRev} :: [\mathit{Int}] \rightarrow \mathit{Bool}$$

$$\mathit{prop\_RevRev}\ \mathit{xs} = $$
$$\quad \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \equiv \mathit{xs}$$

$$\mathit{prop\_RevApp} :: [\mathit{Int}] \rightarrow [\mathit{Int}] \rightarrow \mathit{Bool}$$

$$\mathit{prop\_RevApp}\ \mathit{xs}\ \mathit{ys} = $$
$$\quad \mathit{reverse}\ (\mathit{xs} +\!\!+\ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{ys} +\!\!+\ \mathit{reverse}\ \mathit{xs}$$

$$\mathit{quickCheck}\ (\mathit{prop\_RevRev}\ .\&\&.\ \mathit{prop\_RevApp})$$

Result: `+++ OK, passed 100 tests`

Class *Testable*

Type of quickCheck:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Class *Testable*

Type of quickCheck:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Testable and some instances:

$$\mathbf{class}\ Testable\ prop\ \mathbf{where}$$
$$property\ \ :: prop \rightarrow Property$$
$$exhaustive :: prop \rightarrow Bool$$

$$\mathbf{instance}\ Testable\ Bool$$
$$\mathbf{instance}\ Testable\ Property$$
$$\mathbf{instance}\ (Arbitrary\ a, Show\ a, Testable\ prop) \Rightarrow Testable\ (a \rightarrow prop)$$

Class *Arbitrary*

$$\textbf{class}\ \mathit{Arbitrary}\ a\ \textbf{where}$$
$$\quad \mathit{arbitrary} :: \mathit{Gen}\ a$$
$$\quad \mathit{shrink} :: a \rightarrow [\,a\,]$$

$$\mathit{generate} :: \mathit{Gen}\ a \rightarrow \mathit{IO}\ a$$

Arbitrary instance for all basic types provided. Easy to define additional ones.

Class *Arbitrary*

$$\textbf{class}\ Arbitrary\ a\ \textbf{where}$$
$$\quad arbitrary :: Gen\ a$$
$$\quad shrink :: a \to [\,a\,]$$

$$generate :: Gen\ a \to IO\ a$$

Arbitrary instance for all basic types provided. Easy to define additional ones.

Gen is a *Monad*, *Applicative*, *Functor* (and more).

Class $Arbitrary$

$$
\begin{aligned}
&\textbf{class}\ Arbitrary\ a\ \textbf{where}\\
&\quad arbitrary :: Gen\ a\\
&\quad shrink :: a \rightarrow [a]
\end{aligned}
$$

$$generate :: Gen\ a \rightarrow IO\ a$$

$Arbitrary$ instance for all basic types provided.
Easy to define additional ones.

Gen is a $Monad$, $Applicative$, $Functor$ (and more).

Example:

$$generate\ (arbitrary :: Gen\ [Int])$$

Result: `[28,-2,-26,6,8,8,1]`

Generators (1)

Generators can further be constructed directly for any type in the class $Random$:

$$chooseAny :: Random\ a \Rightarrow Gen\ a$$
$$choose :: Random\ a \Rightarrow (a, a) \rightarrow Gen\ a$$

The latter can be used to state properties that only hold over a specific range.

Generators (2)

Int and any enumeration type are in the class $Random$. The following are efficient specializations of $choose$:

$$chooseEnum :: Enum\ a \Rightarrow (a, a) \rightarrow Gen\ a$$
$$chooseInt :: (Int, Int) \rightarrow Gen\ Int$$

Generators (2)

Int and any enumeration type are in the class $Random$. The following are efficient specializations of $choose$:

$$chooseEnum :: Enum\ a \Rightarrow (a, a) \rightarrow Gen\ a$$
$$chooseInt :: (Int, Int) \rightarrow Gen\ Int$$

Generators can also be constrained by a predicate:

$$suchThat :: Gen\ a \rightarrow (a \rightarrow Bool) \rightarrow Gen\ a$$

Stating Properties (1)

Implication is used to state that a property should hold whenever a precondition is satisfied:

$$(==>) :: Testable\ prop \Rightarrow Bool \rightarrow prop \rightarrow Property$$

Stating Properties (1)

Implication is used to state that a property should hold whenever a precondition is satisfied:

$$(==>) :: \mathit{Testable\ prop} \Rightarrow \mathit{Bool} \rightarrow \mathit{prop} \rightarrow \mathit{Property}$$

For example, the following is a property relating a real (represented by Double) number to its square:

$$\mathit{prop\_SquareLarger} :: \mathit{Double} \rightarrow \mathit{Bool}$$
$$\mathit{prop\_SquareLarger}\ x = x \uparrow 2 > x$$

Stating Properties (2)

It is not universally true, of course:

$$quickCheck\ prop\_SquareLarger$$

Result: `*** Failed! Falsifiable (after 1 test): 0.0`

Stating Properties (2)

It is not universally true, of course:

$$quickCheck\ prop\_SquareLarger$$

Result: `*** Failed! Falsifiable (after 1 test): 0.0`

But a sufficient precondition is that the number is strictly greater than 1. Thus:

$$quickCheck$$
$$(\lambda x \rightarrow (x > 1) ==> prop\_SquareLarger\ x)$$

Result: `+++ OK, passed 100 tests.`

Stating Properties (3)

Alternatively, ***universal quantification*** allows using a generator that only generates valid data:

$$forAll :: (Show\ a, Testable\ prop) \Rightarrow Gen\ a \to (a \to prop) \to Property$$

Stating Properties (3)

Alternatively, ***universal quantification*** allows using a generator that only generates valid data:

$$forAll :: (Show\ a, Testable\ prop) \Rightarrow$$
$$Gen\ a \rightarrow (a \rightarrow prop) \rightarrow Property$$

For example:

$$quickCheck$$
$$(forAll\ (chooseAny\ `suchThat`\ (>1))$$
$$prop\_SquareLarger)$$

Result: `+++ OK, passed 100 tests.`

Stating Properties (4)

A generator that generates valid test data is typically more efficient than generating data and discarding what does not fit. For example:

$$
\begin{aligned}
&prop\_Index :: Eq\ a \Rightarrow [\,a\,] \rightarrow Property \\
&prop\_Index\ xs = \\
&\quad length\ xs > 0 ==> \\
&\qquad forAll\ (choose\ (0, length\ xs - 1))\ \$\ \lambda i \rightarrow \\
&\qquad\quad xs\ !!\ i \equiv head\ (drop\ i\ xs)
\end{aligned}
$$

Note the use of both implication and universal quantification in this partiulcar formulation.

Stating Properties (5)

Properties can be combined using ***conjunction*** and ***disjunction***:

$$(.\&\&.) :: (\mathit{Testable}\ \mathit{prop1}, \mathit{Testable}\ \mathit{prop2}) \Rightarrow \mathit{prop1} \rightarrow \mathit{prop2} \rightarrow \mathit{Property}$$

$$(.||.) :: (\mathit{Testable}\ \mathit{prop1}, \mathit{Testable}\ \mathit{prop2}) \Rightarrow \mathit{prop1} \rightarrow \mathit{prop2} \rightarrow \mathit{Property}$$

Modifiers (1)

A number of newtypes with $Arbitrary$ instances.
E.g. $NonEmptyList\ a$, $SortedList\ a$,
$NonNegative\ a$

Modifiers (1)

A number of newtypes with $Arbitrary$ instances.
E.g. $NonEmptyList\ a$, $SortedList\ a$,
$NonNegative\ a$

Typical definitions:

$$\mathbf{newtype}\ NonEmptyList\ a = NonEmpty\ \{getNonEmpty :: [a]\}$$

$$\mathbf{newtype}\ NonNegative\ a = NonNegative\ \{getNonNegative :: a\}$$

Allows to more precice formulations

Modifiers (2)

Alternative formulation of the index property with a ***type*** that captures that it holds only for non-empty lists (thus avoiding the precondition):

$$
\begin{aligned}
&prop\_Index ::\\
&\quad Eq\ a \Rightarrow NonEmptyList\ a \rightarrow Property\\
&prop\_Index\ (NonEmpty\ xs) =\\
&\quad forAll\ (choose\ (0, length\ xs - 1))\ \$\ \lambda i \rightarrow\\
&\qquad xs\ !!\ i \equiv head\ (drop\ i\ xs)
\end{aligned}
$$

Runnnig Tests

Basic function to run tests:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Runnnig Tests

Basic function to run tests:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Printing of all test cases:

$$verboseCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Runnnig Tests

Basic function to run tests:

$$quickCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Printing of all test cases:

$$verboseCheck :: Testable\ prop \Rightarrow prop \rightarrow IO\ ()$$

Controlling e.g. number and size of test cases:

$$quickCheckWith ::$$
$$\quad Testable\ prop \Rightarrow Args \rightarrow prop \rightarrow IO\ ()$$

$$quickCheckWith$$
$$\quad (stdArgs\ \{ maxSize = 10, maxSuccess = 1000 \})$$
$$\quad prop\_XXX$$

Labelling and Coverage (1)

$label$ attaches a label to a test case:

$$label :: Testable\ prop \Rightarrow String \rightarrow prop \rightarrow Property$$

Example:

$$\begin{array}{l} prop\_RevRev :: [Int] \rightarrow Property \\ prop\_RevRev\ xs = \\ \quad label\ (\texttt{"length is "} +\!\!+\ show\ (length\ xs))\ \$ \\ \qquad reverse\ (reverse\ xs) === xs \end{array}$$

Labelling and Coverage (2)

Result:

```
+++ OK, passed 100 tests:
7% length is 7
6% length is 3
5% length is 4
4% length is 6
```

There are also *cover* and *checkCover* for checking/enforcingig specific coverage requirements.

A Cautionary Tale (1)

$$prop\_Sqrt :: Double \to Bool$$
$$prop\_Sqrt\ x$$
$$\begin{aligned} &\mid x < 0 && = isNaN\ sqrtX \\ &\mid x \equiv 0 \vee x \equiv 1 && = sqrtX \equiv x \\ &\mid x < 1 && = sqrtX > x \\ &\mid x > 1 && = sqrtX > 0 \wedge sqrtX < x \end{aligned}$$

where

$$sqrtX = sqrt\ x$$

$$main = quickCheck\ propSqrt$$

A Cautionary Tale (1)

$$prop\_Sqrt :: Double \rightarrow Bool$$
$$prop\_Sqrt\ x$$
$$\begin{array}{ll} \mid x < 0 & = isNaN\ sqrtX \\ \mid x \equiv 0 \lor x \equiv 1 & = sqrtX \equiv x \\ \mid x < 1 & = sqrtX > x \\ \mid x > 1 & = sqrtX > 0 \land sqrtX < x \end{array}$$

where

$$sqrtX = sqrt\ x$$

$$main = quickCheck\ propSqrt$$

Result: `+++ OK, passed 100 tests`

A Cautionary Tale (2)

$$prop\_Sqrt :: Double \rightarrow Bool$$
$$prop\_Sqrt\ x$$
$$\ldots$$
$$\textbf{where}$$
$$sqrtX = flawedSqrt\ x$$
$$flawedSqrt\ x \mid x \equiv 1 = 0$$
$$\mid otherwise = sqrt\ x$$
$$main = quickCheck\ propSqrt$$

A Cautionary Tale (2)

$$
\begin{aligned}
&prop\_Sqrt :: Double \rightarrow Bool\\
&prop\_Sqrt\ x\\
&\qquad \ldots\\
&\quad \mathbf{where}\\
&\qquad sqrtX = flawedSqrt\ x\\
&\qquad flawedSqrt\ x \mid x \equiv 1 \qquad = 0\\
&\qquad\qquad\qquad\quad \mid otherwise = sqrt\ x\\
&main = quickCheck\ propSqrt
\end{aligned}
$$

Result: `+++ OK, passed 100 tests`

A Cautionary Tale (2)

$$
\begin{array}{l}
prop\_Sqrt :: Double \rightarrow Bool \\
prop\_Sqrt\ x \\
\qquad \ldots \\
\quad \textbf{where} \\
\qquad sqrtX = \mathit{flawedSqrt}\ x \\
\qquad \mathit{flawedSqrt}\ x \mid x \equiv 1 \qquad = 0 \\
\qquad \qquad \qquad \quad \mid otherwise = sqrt\ x \\
main = quickCheck\ propSqrt
\end{array}
$$

Result: `+++ OK, passed 100 tests`

Errr ...

A Cautionary Tale (3)

$$prop\_Sqrt :: Double \rightarrow Bool$$
$$prop\_Sqrt\ x$$
$$\ldots$$
where
$$sqrtX = flawedSqrt\ x$$
$$\ldots$$
$$main = quickCheckWith\ (stdArgs\ \{\, maxSuccess = 1000000\,\})\ propSqrt$$

A Cautionary Tale (3)

$$
\begin{array}{l}
prop\_Sqrt :: Double \rightarrow Bool \\
prop\_Sqrt\ x \\
\qquad \ldots \\
\quad \textbf{where} \\
\qquad sqrtX = flawedSqrt\ x \\
\qquad \ldots \\
main = quickCheckWith \\
\qquad\qquad (stdArgs\ \{ maxSuccess = 1000000 \}) \\
\qquad\qquad propSqrt
\end{array}
$$

Result: `+++ OK, passed 1000000 tests`

A Cautionary Tale (3)

$$
\begin{array}{l}
prop\_Sqrt :: Double \rightarrow Bool \\
prop\_Sqrt\ x \\
\quad\quad \ldots \\
\quad \textbf{where} \\
\quad\quad sqrtX = flawedSqrt\ x \\
\quad\quad \ldots \\
main = quickCheckWith \\
\quad\quad\quad\quad (stdArgs\ \{ maxSuccess = 1000000 \}) \\
\quad\quad\quad\quad propSqrt
\end{array}
$$

Result: `+++ OK, passed 1000000 tests`

Oops. (Very unlikely 1.0 will be picked)

A Cautionary Tale (4)

Simply test specific cases when needed:

$$prop\_Sqrt0 :: Bool$$
$$prop\_Sqrt0 = mySqrt\ 0 \equiv 0$$

$$prop\_Sqrt1 :: Bool$$
$$prop\_Sqrt1 = mySqrt\ 1 \equiv 1$$

A Cautionary Tale (5)

$$
\begin{aligned}
&prop\_SqrtX :: Double \rightarrow Bool\\
&prop\_SqrtX\ x\\
&\quad \mid x < 0\ = isNaN\ sqrtX\\
&\quad \mid x \leqslant 1 = sqrtX \geqslant x\\
&\quad \mid x > 1\ = sqrtX > 0 \wedge sqrtX < x\\
&\quad \textbf{where}\\
&\qquad sqrtX = mySqrt\ x
\end{aligned}
$$

A Cautionary Tale (6)

$$
\begin{aligned}
&\mathit{prop\_Sqrt} :: \mathit{Property}\\
&\mathit{prop\_Sqrt} = \mathit{counterexample}\\
&\qquad\qquad \texttt{"sqrt 0 failed"}\\
&\qquad\qquad \mathit{prop\_Sqrt0}\\
&\quad .\&\&. \quad \mathit{counterexample}\\
&\qquad\qquad \texttt{"sqrt 1 failed"}\\
&\qquad\qquad \mathit{prop\_Sqrt1}\\
&\quad .\&\&. \quad \mathit{prop\_SqrtX}
\end{aligned}
$$

(*counterexample* adds a string to a property that gets printed if the property fails.)

Testing Interval Arithmetic (1)

Lifting a unary operator $\ominus$ to an operator $\hat{\ominus}$ working on intervals is defined as follows, assuming $\ominus$ is defined on the entire interval:

$$\hat{\ominus} i = [\min_{\forall x \in i} \ominus x, \ \max_{\forall x \in i} \ominus x]$$

Testing Interval Arithmetic (1)

Lifting a unary operator $\ominus$ to an operator $\hat{\ominus}$ working on intervals is defined as follows, assuming $\ominus$ is defined on the entire interval:

$$\hat{\ominus} i = [\min_{\forall x \in i} \ominus x, \ \max_{\forall x \in i} \ominus x]$$

And for binary operators:

$$i_1 \mathbin{\hat{\otimes}} i_2 = [\min_{\forall x \in i_1, y \in i_2} x \otimes y, \ \max_{\forall x \in i_1, y \in i_2} x \otimes y]$$

Testing Interval Arithmetic (2)

But how can we test that? In general, very difficult to find the global minimum/maximum of a function over an interval without further information e.g. about its derivatives.

Testing Interval Arithmetic (2)

But how can we test that? In general, very difficult to find the global minimum/maximum of a function over an interval without further information e.g. about its derivatives.

However, for a given interval i, it follows that:

$$\forall x \in i. \ominus x \in \hat{\ominus} i$$

Testing Interval Arithmetic (3)

Unfortunately, $\hat{\ominus} i = [-\infty, +\infty]$ satisfies

$$\forall x \in i.\ \ominus x \in \hat{\ominus} i$$

Testing Interval Arithmetic (3)

Unfortunately, $\hat{\ominus} i = [-\infty, \ +\infty]$ satisfies

$$\forall x \in i.\, \ominus\, x \in \hat{\ominus} i$$

We should ideally test that the result interval is not larger than necessary. But that is hard too.

Testing Interval Arithmetic (3)

Unfortunately, $\hat{\ominus} i = [-\infty,\ +\infty]$ satisfies

$$\forall x \in i.\, \ominus\, x \in \hat{\ominus} i$$

We should ideally test that the result interval is not larger than necessary. But that is hard too.

However, the definition does imply that a 1-point interval must be mapped to a 1-point interval:

$$\hat{\ominus}[x, x] = [\ominus x,\ \ominus x]$$

While not perfect, does rule out trivial implementations and it is easy to test.

Testing Interval Arithmetic (4)

For binary operators:

- For given intervals i_1 and i_2:

$$\forall x \in i_1, y \in i_2.\ x \otimes y \in i_1 \hat{\otimes} i_2$$

- For given x and y:

$$[x, x] \,\hat{\otimes}\, [y, y] = [x \otimes y,\ x \otimes y]$$

Let us turn the above into QuickCheck test cases interactively. (2021: Exercise!)