QuickCheck is a framework for automatic program testing:

- Allows to test whether the program satisfies given properties on randomly generated inputs.
- originally for Haskell, but now also for Erlang, Scala, Java, etc.
- The library has:
  - combinators for describing properties which functions should satisfy;
  - generators of random values for standard types;
  - combinators for constructing generators for user defined datatypes.

Basic interface

```
import Test.QuickCheck

quickCheck :: Testable prop => prop -> IO ()
```

- Function `quickCheck` gets a property as an argument which is tested on randomly generated inputs.
- By default 100 times (but this is configurable).
- If some tests fail it outputs a counter example.

Example

Properties of `reverse`

$$\forall z.\text{reverse}\ (\text{reverse}\ z) = z$$

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

$$\forall a.\forall b.\text{reverse}\ (a{+}{+}b) = (\text{reverse}\ b){+}{+}(\text{reverse}\ a)$$

```
prop_RevApp           ::  [Int] -> [Int] -> Bool
prop_RevApp xs ys     =   reverse (xs ++ ys)
                          == reverse ys ++ reverse xs
```

Example

Properties of `reverse`

$$\forall z.\text{reverse}\ (\text{reverse}\ z) = z$$

Let's try it out ...

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

Main> quickCheck prop_RevApp
+++ OK, passed 100 tests.
```

```
prop_RevApp           ::  [Int] -> [Int] -> Bool
prop_RevApp xs ys     =   reverse (xs ++ ys)
                          == reverse ys ++ reverse xs
```

Example (cont.)

Properties of `reverse`

$$\forall a.\forall b.\text{reverse } (a{+}{+}b) = (\text{reverse } b){+}{+}(\text{reverse } a)$$

```
prop_RevApp           ::  [Int] -> [Int] -> Bool
prop_RevApp xs ys     =   reverse (xs ++ ys)
                          == reverse ys ++ reverse xs
```

$$\forall a.\forall b.\text{reverse } (a{+}{+}b) = (\text{reverse } a){+}{+}(\text{reverse } b)$$

```
prop_RevWrong         ::  [Int] -> [Int] -> Bool
prop_RevWrong xs ys   =   reverse (xs ++ ys)
                          == reverse xs ++ reverse ys
```

Example (cont.)

Properties of `reverse`

$$\forall a.\forall b.\text{reverse } (a{+\!+}b) = (\text{reverse } b){+\!+}(\text{reverse } a)$$

Let's try it out ...

```
Main> quickCheck prop_RevWrong
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
[0]
[1]
```

```
prop_RevWrong          ::  [Int] -> [Int] -> Bool
prop_RevWrong xs ys    =   reverse (xs ++ ys)
                           == reverse xs ++ reverse ys
```

Properties

```
class Testable prop where
      property :: prop -> Property

instance Testable Bool

instance (Arbitrary a, Show a, Testable prop) =>
         Testable (a -> prop)
```

- Properties are expressions which type belongs to class `Testable`.
- Arguments should be of a monomorphic type.
  - Necessary for knowing how to generate arguments.
- Naming convention: prefix `prop_`

Properties

Example: insertion sort

```
isort :: Ord a => [a] -> [a]
isort = foldr insert []

insert :: Ord a => a -> [a] -> [a]
insert x []                    = [x]
insert x (y:ys) | x <= y       = x : y : ys
                | otherwise = y : insert x ys
```

Properties

Sorting property 1: sorted list must be ordered

```
prop_sortOrder :: [Int] -> Bool
prop_sortOrder xs = ordered (isort xs)

ordered :: Ord a => [a] -> Bool
ordered (x:y:ys)  = x <= y && ordered (y:ys)
ordered ys        = True
```

Properties

Sorting property 2: sorted and original list have same elements

```
prop_sortElems :: [Int] -> Bool
prop_sortElems xs = sameElems xs (isort xs)

sameElems :: Eq a => [a] -> [a] -> Bool
sameElems xs ys   = null (xs \\ ys) && null (ys \\ xs)
```

Properties

Inspecting test data

```
collect :: (Show a, Testable prop) => a -> prop -> Property
```

- Function `collect` gathers statistics about test cases.
- This information is displayed when a test passes.

Properties

How many test cases were non-empty?

```
Main> let p = prop_sortOrder
Main> quickCheck (\ xs -> collect (null xs) (p xs))
+++ OK, passed 100 tests.
93% False
 7% True
```

Properties

How long were argument lists?

```
Main> let l20 xs = length xs `div` 20
Main> quickCheck (\xs -> collect (l20 xs) (p xs))
+++ OK, passed 100 tests:
53% 0
22% 1
14% 2
 7% 3
 4% 4
```

Properties

What were actual arguments?

```
Main> quickCheck (\ xs -> collect xs (p xs))
+++ OK, passed 100 tests:
 8% []
 1% [97723,95805,-104521,45943,-73844,6249,64936]
...
```

Properties

Insertion property: insertion preserves sorting (ver. 1)

```
prop_insertOrder1       :: Int -> [Int] -> Bool
prop_insertOrder1 x xs =  ordered xs `implies`
                                ordered (insert x xs)

implies      :: Bool -> Bool -> Bool
implies x y =  not x || y
```

Properties

Insertion property: insertion preserves sorting

Problem!

```
Main> let p = prop_insertOrder1
Main> quickCheck(\x xs -> collect(ordered xs)(p x xs))
+++ OK, passed 100 tests:
87% False
13% True
```

```
implies x y =  not x || y
```

Properties

Implications

```
(==>) :: Testable prop => Bool -> prop -> Property

instance Testable Property
```

- The combinator `(==>)` ignores inputs where premise is not satisfied and regenerates new test data.
- By default 500 times (but this is configurabe).

Properties

Insertion property: insertion preserves sorting (ver. 2)

```
prop_insertOrder2        :: Int -> [Int] -> Property
prop_insertOrder2 x xs =  ordered xs ==>
                               ordered (insert x xs)
```

Properties

Insertion property: insertion preserves sorting

(

p

p

Better but still . . .

```
Main> let p = prop_insertOrder2
Main> quickCheck(\x xs -> collect(ordered xs)(p x xs))
*** Gave up! Passed only 82 tests (100% True).
```

Properties

Universal quantification

```
forAll :: (Show a, Testable prop) =>
                  Gen a -> (a -> prop) -> Property
```

- The combinator `forAll` gets an explicit generator which is used for generating test cases.
- Allows to use special generators which guarantee that input satisfies certain properties.

Properties

Insertion property: insertion preserves sorting (ver. 3)

```
prop_insertOrder3    :: Int -> Property
prop_insertOrder3 x =  forAll orderedList (\ xs ->
                                  ordered (insert x xs))
```

Properties

In
(v
pr
pr

Now works!!

```
Main> quickCheck (forAll orderedList ordered)
+++ OK, passed 100 tests.
Main> quickCheck prop_insertOrder3
+++ OK, passed 100 tests.
```

ts))

Generators

Generators

```
newtype Gen a = ...

instance Monad Gen

instance Functor Gen

instance (Testable prop) => Testable (Gen prop)
```

- Generators belong to an abstract data type `Gen`.
- `Gen` is a monad which effect is ”access” to random numbers.

Generators

Sampling generated data

```
sample :: Show a => Gen a -> IO ()
```

Combinators for generators

```
choose      :: Random a => (a, a) -> Gen a
elements    :: [a] -> Gen a
oneof       :: [Gen a] -> Gen a
frequency   :: [(Int, Gen a)] -> Gen a
sized       :: (Int -> Gen a) -> Gen a
vectorOf    :: Int -> Gen a -> Gen [a]
```

Generators

Default generators

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

  shrink _  =  []
```

- Types belonging to the class `Arbitrary` have the default generator.
- In addition, the class has a method `shrink` which is used for generating smaller counterexamples:
  - `shrink` returns a list of structurally smaller values;
  - if the property fails it is retested on values returned by `shrink` until there is no smaller counterexamples.

Generators

Simple generators

```
instance Arbitrary Bool where
  arbitrary  = choose (False,True)

instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
  arbitrary  = liftM2 (,) arbitrary arbitrary


data Color = Red | Blue | Green

instance Arbitrary Color where
  arbitrary  = elements [Red, Blue, Green]
```

Generators

Simple generators

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = oneof [ return Nothing
                    , liftM Just arbitrary]
```

Generators

Simple generators

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = oneof [ return Nothing
```

Problem!

Half of the values are `Nothing`!!

Generators

Simple generators

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = oneof [ return Nothing
                    , liftM Just arbitrary]
```

A better version

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = frequency [ (1, return Nothing)
                        , (3, liftM Just arbitrary)]
```

Generators

Generating integers (ver. 1)

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

Generators

Generating integers (ver. 1)

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

Generating integers (ver. 2)

```
instance Arbitrary Int where
  arbitrary = sized (\ n -> choose (-n,n))
```

Generators

Generating recursive data types (ver. 1)

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

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency [(1, liftM  Leaf arbitrary)
                        ,(2, liftM2 Node arbitrary arbitrary)]
```

Generators

Generating recursive data types (ver. 1)

```
data Tree a  =  Leaf a
```

Problem!

Termination is not guaranteed!!

```
insta
  arbitrary = frequency [(1, liftM  Leaf arbitrary)
                        ,(2, liftM2 Node arbitrary arbitrary)]
```

Generators

Generating recursive data types (ver. 2)

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbitraryTree

arbitraryTree    :: Arbitrary a => Int -> Gen (Tree a)
arbitraryTree 0 =  liftM Leaf arbitrary
arbitraryTree n =  frequency [ (1, liftM  Leaf arbitrary)
                             , (4, liftM2 Node t t)]
  where t = arbitraryTree (n `div` 2)
```

NB!

- The second equation has the possibility to generate `Leaf`.
- Otherwise would generate only balanced trees.

Generators

Predefined special generators

```
newtype OrderedList a  = Ordered [a]
instance (Ord a, Arbitrary a) => Arbitrary (OrderedList a)

newtype NonEmptyList a = NonEmpty [a]
instance Arbitrary a => Arbitrary (NonEmptyList a)

newtype Positive a     = Positive a
instance (Num a, Ord a, Arbitrary a) => Arbitrary (Positive a)

newtype NonZero a      = NonZero a
newtype NonNegative a  = NonNegative a
```

Function Generators

```
class CoArbitrary a where
  coarbitrary :: a -> Gen b -> Gen b

instance (CoArbitrary a, Arbitrary b) => Arbitrary (a -> b)
```

Example

```
variant :: Integral n => n -> Gen a -> Gen a

instance CoArbitrary a => CoArbitrary [a] where
  coarbitrary []     = variant 0
  coarbitrary (x:xs) = variant 1 . coarbitrary (x,xs)
```

- You should use `variant` to perturb the random generator; the goal is that different values for the first argument will lead to different calls to `variant`.

QuickCheck

Conclusion

- As Haskell is lazy language, it allows to use infinite values; but properties may inspect only a finite part of it.
- Also provides features to test monadic values (incl. `IO`).