**Property Testing with QuickCheck**

Functional Programming

Utrecht University

- Gain confidence in the correctness of your program
- Show that common cases work correctly
- Show that *corner* cases work correctly

- Gain confidence in the correctness of your program
- Show that common cases work correctly
- Show that *corner* cases work correctly

**Testing cannot prove the absence of bugs**

# When is a program correct?

## When is a program correct?

- When it satisfies the specification

## When is a program correct?

- When it satisfies the specification

- What is a specification?
- How to establish a relation between the specification and the implementation?
- What about bugs in the specification?

More in *Software Testing and Verification*, period 4

QuickCheck, an *automated* testing library/tool for Haskell

- Describe properties as Haskell programs using an embedded domain-specific language (EDSL)
- Automatic datatype-driven random test case generation
- Extensible, e.g. test case generators can be adapted
    - A default generator for list generates any list, but you may want only sorted lists

**Case study: insertion sort**

## A buggy insertion sort

```
sort :: [Int] -> [Int]
sort []      =  []
sort (x:xs)  =  insert x xs

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

Let's try to debug it using QuickCheck

A good specification is

- as precise as necessary
- but no more precise than necessary

A good specification for a particular problem, such as sorting, should:

1. distinguish sorting from all other operations on lists,
2. without forcing us to use a particular sorting algorithm

## A first approximation

Certainly, sorting a list should not change its length

```haskell
sortPreservesLength :: [Int] -> Bool
sortPreservesLength xs =
  length (sort xs) == length xs
```

## A first approximation

Certainly, sorting a list should not change its length

```
sortPreservesLength :: [Int] -> Bool
sortPreservesLength xs =
  length (sort xs) == length xs
```

We can test by invoking the function:

```
> quickCheck sortPreservesLength
Failed! Falsifiable, after 4 tests:
[0,3]
```

QuickCheck gives back a *counterexample*

## Correcting the bug

```haskell
sort :: [Int] -> [Int]
sort []     =  []
sort (x:xs) =  insert x xs

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

Which branch does not preserve the list length?

## Correcting the bug

```haskell
sort :: [Int] -> [Int]
sort []     =  []
sort (x:xs) =  insert x xs

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

```
> quickCheck sortPreservesLength
OK, passed 100 tests.
```

Looks better. But have we tested enough?

## A different "sorting" algorithm....

```haskell
sortPreservesLength :: [Int] -> Bool
sortPreservesLength xs =
  length (sort xs) == length xs

idPreservesLength :: [Int] -> Bool
idPreservesLength xs =
  length (id xs) == length xs
```

12

## A different "sorting" algorithm....

```haskell
sortPreservesLength :: [Int] -> Bool
sortPreservesLength xs =
  length (sort xs) == length xs

idPreservesLength :: [Int] -> Bool
idPreservesLength xs =
  length (id xs) == length xs

> quickCheck idPreservesLength
OK, passed 100 tests.
```

So we need to refine our specification

## First, some refactoring ..

```
preserves                 :: Eq b => (a -> a) -> (a -> b) -> a -> Bool
(algo `preserves` prop) x = prop (algo x) == prop x
```

## First, some refactoring ..

```haskell
preserves                :: Eq b => (a -> a) -> (a -> b) -> a -> Bool
(algo `preserves` prop) x = prop (algo x) == prop x

sortPreservesLength  =  sort `preserves` length
idPreservesLength    =  id   `preserves` length
```

**When is a list sorted?**

We can define a predicate that checks if a list is sorted:

```
isSorted            :: [Int] -> Bool
isSorted []         = True
isSorted [x]        = True
isSorted (x:y:xs)   = x < y && isSorted (y:xs)
```

And use this to check that sorting a list produces a list that isSorted

**Testing again**

```
sortEnsuresSorted    :: [Int] -> Bool
sortEnsuresSorted xs = isSorted (sort xs)

> quickCheck sortEnsuresSorted
Falsifiable, after 5 tests:
[5,0,-2]
> sort [5,0,-2]
[0,-2,5]
```

We're still not quite there…

What's wrong now?

```
sort       :: [Int] -> [Int]
sort []    =  []
sort (x:xs) =  insert x xs

insert :: Int -> [Int] -> [Int]
```

## Debugging sort

What's wrong now?

```
sort       :: [Int] -> [Int]
sort []    =  []
sort (x:xs) =  insert x xs

insert :: Int -> [Int] -> [Int]
```

We are not recursively sorting the tail in sort!

## Debugging sort

What's wrong now?

```haskell
sort :: [Int] -> [Int]
sort []     =  []
sort (x:xs) =  insert x (sort xs)

insert :: Int -> [Int] -> [Int]
```

## Another bug

```
> quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
[4,2,2]
> sort [4,2,2]
[2,2,4]
```

This is correct. What is wrong?

## Another bug

```
> quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
[4,2,2]
> sort [4,2,2]
[2,2,4]
```

This is correct. What is wrong?

```
> isSorted [2,2,4]
False
```

The isSorted specification reads:

```
sorted          :: [Int] -> Bool
sorted []       =  True
sorted (x:[])   =  True
sorted (x:y:ys) =  x < y && sorted (y : ys)
```

Why does it return False? How can we fix it?

## Fixing the specification

The isSorted specification reads:

```
sorted :: [Int] -> Bool
sorted []        =  True
sorted (x:[])    =  True
sorted (x:y:ys)  =  x <= y && sorted (y : ys)
```

Is sorting specified completely by saying that
- sorting preserves the length of the input list,
- the resulting list is sorted?

## Are we done yet?

Is sorting specified completely by saying that

- sorting preserves the length of the input list,
- the resulting list is sorted?

Not really...

```haskell
evilNoSort :: [Int] -> [Int]
evilNoSort xs = replicate (length xs) 1
```

This function fulfills both specifications, but does not sort

## Specifying sorting

```
permutes :: ([Int] -> [Int]) -> [Int] -> Bool
permutes f xs = f xs `elem` permutations xs

sortPermutes :: [Int] -> Bool
sortPermutes xs = sort `permutes` xs
```

This completely specifies sorting and our algorithm passes the corresponding tests:

```
sorts        :: ([Int] -> [Int]) -> [Int] -> Bool
sorts alg xs = and [ alg `permutes` xs
                   , alg `preserves` length
                   , sorted (alg xs)
                   ]
```

Are we now done?

## A minimal specification

This completely specifies sorting and our algorithm passes the corresponding tests:

```
sorts        :: ([Int] -> [Int]) -> [Int] -> Bool
sorts alg xs = and [ alg `permutes` xs
                   , sorted (alg xs)
                   ]
```

# QuickCheck in general

## The type of `quickCheck`

The type of is an *overloaded* type:

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

- The argument of is a property of type prop
- The only restriction on the type is that it is in the Testable *type class*.
- When executed, prints the results of the test to the screen – hence the IO () result type.

## Which properties are `Testable`?

So far, all our properties have been of type:

```
sortPreservesLength :: [Int] -> Bool
sortEnsuresSorted :: [Int] -> Bool
sortPermutes :: [Int] -> Bool
```

When used on such properties, QuickCheck generates random integer lists:
- If the result is `True` for 100 cases, this success is reported in a message
- If the result is `False` for a test case, the input triggering the failure is printed

## Nullary properties

A property without arguments is also possible:

```
lengthEmpty :: Bool
lengthEmpty = length [] == 0

wrong :: Bool
wrong = False

> quickCheck lengthEmpty
OK, passed 100 tests.
> quickCheck wrong
Falsifiable, after 0 tests.
```

**QuickCheck subsumes unit tests**

## Other example properties

```
appendLength :: [Int] -> [Int] -> Bool
appendLength xs ys =
  length xs + length ys == length (xs ++ ys)

plusIsCommutative :: Int -> Int -> Bool
plusIsCommutative m n = m + n == n + m

takeDrop :: Int -> [Int] -> Bool
takeDrop n xs = take n xs ++ drop n xs == xs

dropTwice :: Int -> Int -> [Int] -> Bool
dropTwice m n xs =
  drop m (drop n xs) == drop (m + n) xs
```

## Other forms of properties – contd.

```
> quickCheck takeDrop
OK, passed 100 tests.


> quickCheck dropTwice
Falsifiable after 7 tests.
1
-1
[0]


> drop (-1) [0]
[0]


> drop 1 (drop (-1) [0])
[]
```

Recall the type of quickCheck:

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

We can now say more about when types are Testable:

- testable properties usually are functions (with any number of arguments) resulting in a Bool

What argument types are admissible?

## Properties

Recall the type of quickCheck:

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

We can now say more about when types are Testable:

- testable properties usually are functions (with any number of arguments) resulting in a Bool

What argument types are admissible?

- QuickCheck has to know how to produce random test cases of such types

## Properties – continued

A `Testable` thing is something which can be turned into a `Property`:

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

A Bool is testable:

```
instance Testable Bool where ...
```

If a type is testable, we can add a function argument, as long as we know how to generate and print test cases:

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

## Information about test data

We can show the actual data that is tested:

```
> quickCheck (\xs -> collect xs (sorts sort xs))
OK, passed 100 tests:
6% []
1% [9,4,-6,7]
1% [9,-1,0,-22,25,32,32,0,9,...
...
```

Why is it important to have access to the test data?

The function insert preserves an ordered list:

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

insertPreservesOrdered :: Int -> [Int] -> Bool
insertPreservesOrdered x xs = sorted xs `implies` sorted (insert x xs)
```

# Implications – contd.

```
> quickCheck insertPreservesOrdered
OK, passed 100 tests.
```

**Implications – contd.**

```
> quickCheck insertPreservesOrdered
OK, passed 100 tests.

But:

> quickCheck (\x xs -> collect (sorted xs)
                               (insertPreservesOrdered x xs))
OK, passed 100 tests.
88% False
12% True
```

For **88** test cases, insert has not actually been relevant!

The solution is to use the QuickCheck implication operator:

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

insertPreservesOrdered     :: Int -> [Int] -> Property
insertPreservesOrdered x xs = sorted xs ==> sorted (insert x xs)
```

Now, lists that are not sorted are discarded and do not contribute towards the goal of 100 test cases

## Implications – contd.

We can now easily run into a new problem:

```
insertPreservesOrdered    :: Int -> [Int] -> Property
insertPreservesOrdered x xs =
    length xs > 2 && sorted xs ==> sorted (insert x xs)
```

We try to ensure that lists are not too short, but:

```
> quickCheck (\x xs -> collect (sorted xs)
                               (insertPreservesOrdered x xs))
Arguments exhausted after 20 tests (100% True).
```

The chance that a random list is sorted is extremely small

# Custom generators

- Generators belong to an abstract data type Gen
  - The only effect available to us is access to random numbers
  - Think of as a restricted version of IO
- We can define our own generators using another domain-specific language
  - The default generators for datatypes are specified by defining instances of class Arbitrary

  ```
  class Arbitrary a where
    arbitrary :: Gen a

    ...
  ```

## Generator combinators

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

## Simple generators

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


instance (Arbitrary a, Arbitrary b)
      => Arbitrary (a,b) where
  arbitrary =  do x <- arbitrary
                  y <- arbitrary
                  return (x,y)
  -- arbitrary = (,) <$> arbitrary <*> arbitrary


data Dir = North | East | South | West
instance Arbitrary Dir where
  arbitrary = elements [North, East, South, West]
```

- A simple possibility:

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

- Better:

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

- QuickCheck automatically increases the size gradually

## How to generate sorted lists

**Idea: Adapt the default generator for lists**

The following function turns a list of integers into a sorted list of integers:

```haskell
mkSorted :: [Int] -> [Int]
mkSorted []       = []
mkSorted [x]      = [x]
mkSorted (x:y:ys) = x : mkSorted ((x + abs y : ys))
```

For example:

```
> mkSorted [1,2,-3,4]
[1,3,6,10]
```

## Random generator

The generator can be adapted as follows:

```haskell
genSorted :: Gen [Int]
genSorted = do xs <- arbitrary
               return (mkSorted xs)
-- genSorted = mkSorted <$> arbitrary
```

## Using a custom generator

There is another function to construct properties provided by QuickCheck, passing an explicit generator:

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

This is how we use it:

```
insertPreservesOrdered   :: Int -> Property
insertPreservesOrdered x = forAll genSorted (\xs ->
    length xs > 2 && sorted xs ==> sorted (insert x xs))
```

## Using a custom generator (using a newtype)

Alternatively, use a newtype to define a custom generator:

```
newtype SortedIntList = SortedIntList [Int]

instance Arbitrary SortedIntList where
  arbitrary = genSorted
```

use it as follows:

```
insertCorrectProp                            :: Int -> SortedIntList -> Bool
insertCorrectProp x (SortedIntList xs) = isSorted (insert x xs)
```

**Shrinking**

The other method in `Arbitrary` is:

```
shrink :: (Arbitrary a) => a -> [a]
```

- Maps each value to structurally smaller values
    - `[2,3]` is structurally smaller than `[1,2,3]`
- When a failing test case is discovered, QuickCheck shrinks repeatedly until no smaller failing test case can be obtained

## Loose ends

- Haskell can deal with infinite values, and so can QuickCheck
    - Properties must *not* inspect infinitely many values
    - Solution: only inspect finite parts
- QuickCheck can also generate functional values
    - Tequires defining an instance of another class `Coarbitrary`
    - Showing functional values is still problematic
- QuickCheck has facilities for testing properties that involve `IO`

## Summary

QuickCheck is a great tool:

- A domain-specific language for writing properties
- Test data is generated automatically and randomly
- Another domain-specific language to write custom generators

However, keep in mind that writing good tests still requires practice, and that tests can have bugs, too

**Correctness**

Testing can**not** prove the absence of bugs

- Only point at failing cases

**Are there ways to prove your code correct?**

## Equational reasoning

1. Write a bunch of properties that specify your algorithm
2. Prove that they hold using equational reasoning
3. You are done!

## Equational reasoning

1. Write a bunch of properties that specify your algorithm
2. Prove that they hold using equational reasoning
3. You are done!

### Caveats

- Time-consuming, needs lots of manual work
- Laziness and exceptions are not taken care of
  - Proofs only work for finite values

Help you proving properties about your program

- Check that every inference step is correct
- Fill in boring and obvious proofs

Some interactive theorem provers:

- Coq (blame the French for the name!)
- Isabelle/HOL

## More expressive types

Define the type of your function in such a way that only correct implementations are allowed

```
append :: List n a -> List m a -> List (n + m) a
```

1. Dependent types
    - Allow values to appear in types
    - Examples: Agda, Idris, Coq
2. Refinement types
    - Attach predicates to types
    - Example: LiquidHaskell

## More expressive types

Define the type of your function in such a way that only correct implementations are allowed

```
append :: List n a -> List m a -> List (n + m) a
```

1. Dependent types
   - Allow values to appear in types
   - Examples: Agda, Idris, Coq
2. Refinement types
   - Attach predicates to types
   - Example: LiquidHaskell

**Learn about them in *Advanced FP*!**

## Theorems for free

How many implementations are of these signatures?

```
f :: a -> a
g :: (a, b) -> (b, a)
```

How many implementations are of these signatures?

```
f :: a -> a
g :: (a, b) -> (b, a)
```

Only one!

```
f x     = x       -- identity function
g (x, y) = (y, x) -- swap pair
```

Types are enough to determine many properties of the implementation

- We call those *free theorems*