Advanced Functional Programming

02 - Testing

Wouter Swierstra & Trevor L. McDonell

Utrecht University

Testing and correctness

- When is a program correct?

Testing and correctness

- When is a program correct?

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

Equational reasoning

- “Equals can be substituted for equals”
- In other words: if an expression has a value in a context, we can replace it with any other expression that has the same value in the context without affecting the meaning of the program.
- When we deal with infinite structures: two things are equivalent if we cannot find out about their difference:

Equational reasoning

- “Equals can be substituted for equals”
- In other words: if an expression has a value in a context, we can replace it with any other expression that has the same value in the context without affecting the meaning of the program.
- When we deal with infinite structures: two things are equivalent if we cannot find out about their difference:

```
ones   = 1:   ones
ones'  = 1:1: ones'
```

Referential transparency

In most functional languages like ML or OCaml, there is no referential transparency:

```
let  val  x    =  ref 0
     fun  f n  =  (x := !x + n; !x)
in   f 1 + f 2
```

Referential transparency

In most functional languages like ML or OCaml, there is no referential transparency:

```
let  val  x    =  ref 0
     fun  f n  =  (x := !x + n; !x)
in   f 1 + f 2
```

But we cannot replace the last line with `1 + f 2`, even though `f 1 = 1`.

Referential transparency in Haskell

- Haskell is referentially transparent – all side-effects are tracked by the IO monad.

```haskell
do
  x  <-  newIORef 0
  let f n = do modifyIORef x (+n); readIORef x
  r  <-  f 1
  s  <-  f 2
  return (r + s)
```

Note that the type of `f` is `Int → IO Int` – we cannot safely make the substitution we proposed previously.

Referential transparency

Because we can safely replace equals for equals, we can *reason* about our programs – this is something you already saw in the course on functional programming.

For example to prove some statement `P xs` holds for all lists `xs`, we need to show:

- `P []` – the base case;
- for all `x` and `xs`, `P xs` implies `P (x:xs)`.

Equational reasoning

- Equational reasoning can be an elegant way to prove properties of a program.
- Equational reasoning can be used to establish a relation between an "obviously correct" Haskell program (a specification) and an efficient Haskell program.
- Equational reasoning can become quite long...
- Careful with special cases (laziness):
  - undefined values;
  - partial functions;
  - infinite values.

You can formalize such proofs in other systems such as Agda, Coq or Isabelle.

QuickCheck

QuickCheck, an automated testing library/tool for Haskell

Features:

- 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.

History

- Developed in 2000 by Koen Claessen and John Hughes.
- Copied to other programming languages: Common Lisp, Scheme, Erlang, Python, Ruby, SML, Clean, Java, Scala, F#
- Erlang version is sold by a company, QuviQ, founded by the authors of QuickCheck.

Case study: insertion sort

```haskell
isort :: Ord a ⇒ [a] → [a]
isort []      =  []
isort (x:xs)  =  insert x (isort xs)


insert :: Ord a ⇒ a → [a] → [a]
insert x []      =  [x]
insert x (y:ys)
  | x ⩽ y        =  x : y : ys
  | otherwise    =  y : insert x ys
```

Properties of insertion sort

We can now try to prove that for all lists `xs`,

```
length (sort xs) == length xs.
```

- The base case is trivial.
- The inductive case requires a lemma relating `insert` and `length` – suggestions?

Case study: insertion sort

Consider the following (buggy) implementation of insertion sort:

```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
```

Let’s try to debug it using QuickCheck.

How to write a specification?

A good specification is

- as precise as necessary,
- no more precise than necessary.

A good specification for a particular problem, such as sorting, should distinguish sorting from all other operations on lists, without forcing us to use a particular sorting algorithm.

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]
```

Correcting the bug

```
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?

A new attempt

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

Looks better. But have we tested enough?

Properties are first-class objects

```
(f `preserves` p) x = p x == p (f x)

sortPreservesLength  =  sort `preserves` length

idPreservesLength    =  id `preserves` length
```

Properties are first-class objects

```
(f `preserves` p) x = p x == p (f x)

sortPreservesLength  =  sort `preserves` length

idPreservesLength    =  id `preserves` length
```

So `id` also preserves the lists length:

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

We need to refine our spec.

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

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

We're still not quite there...

Debugging sort

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`.

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
```

Fixing the spec

The `isSorted` spec 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?

Are we done yet?

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?

No, not quite.

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

This function fulfills both specifications, but still does not sort.

We need to make the relation between the input and output lists precise: both should contain the same elements – or one should be a permutation of the other.

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.

How to use QuickCheck

To use QuickCheck in your program:

```haskell
import Test.QuickCheck
```

Define properties.

Then call to test the properties.

```haskell
quickCheck :: Testable prop ⇒ prop → IO ()
```

The type of `quickCheck`

The type of is an *overloaded* type:

```
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 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 and verifies that the result is `True`.

If the result is for 100 cases, this success is reported in a message.

If the result is `False` for a test case, the input triggering the result is printed.

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])
[]
```

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 vs unit tests

No random test cases are involved for nullary properties.

QuickCheck subsumes unit tests.

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

```haskell
class Testable prop where
  property :: prop → Property

instance Testable Bool where
      ...
instance  (Arbitrary a, Show a, Testable b) ⇒
           Testable (a → b) where
```

We can test any Boolean value or any testable function for which we can generate arbitrary input.

More information about test data

```
collect :: (Testable prop, Show a) ⇒
  a → prop → Property
```

The function gathers statistics about test cases. This information is displayed when a test passes:

```
> let sPL = sortPreservesLength
> quickCheck (\xs -> collect (null xs) (sPL xs))
OK, passed 100 tests.
96% False
4% True.
```

The result implies that not all test cases are distinct.

More information about test data – contd.

```
> quickCheck (\xs -> collect (length xs `div` 10)
                              (sPL xs))
+++ OK, passed 100 tests.
 26% 0.
 21% 1.
 15% 2.
 10% 5.
 10% 3.
  ...
```

Most lists are small in size: QuickCheck generates small test cases first, and increases the test case size for later tests.

More information about test data (contd.)

In the extreme case, we can show the actual data that is tested:

```
> quickCheck (\ xs → collect xs (sPL 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?

Implications

The function insert preserves an ordered list:

```
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.
```

But:

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

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

Implications – contd.

The solution is to use the QuickCheck implication operator:

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

```
instance Testable Property
```

The type allows to write a logically equivalent formula that also explicitly rejects the test case.

```
iPO :: Int -> [Int] -> Property
iPO 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:

```
iPO :: Int → [Int] → Property
iPO 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)
                               (iPO x xs))
Arguments exhausted after 20 tests (100% True).
```

The chance that a random list is sorted is extremely small. QuickCheck will give up after a while if too few test cases pass the precondition.

Configuring QuickCheck

```haskell
quickCheckWith :: Testable prop ⇒
  Args → prop → IO ()

data Args where
  replay :: Maybe (StdGen, Int)
  -- should we replay a previous test?
  maxSuccess :: Int
  -- max number of successful tests
  --   before succeeding
  maxDiscardRatio :: Int
  -- max number of discarded tests
  --   per successful test
  maxSize :: Int
  --max test case size
```

Generators

- Instead of increasing the number of test cases to generate, it is usually better to write a custom random generator.
- Generators belong to an abstract data type `Gen`. Think of as a restricted version of `IO`. The only effect available to us is access to random numbers.
- 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

```haskell
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)

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

Generating random numbers

- 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, up to the configured maximum value.

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:

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

Using a custom generator

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

```haskell
forAll :: (Show a, Testable b) ⇒
  Gen a → (a → b) → Property
```

This is how we use it:

```haskell
iPO :: Int → Property
iPO x = forAll genSorted
          (\ xs → length xs > 2 && sorted xs ⟹
                     sorted (insert x xs))
```

Loose ends: Shrinking

`Arbitrary` revisited

```haskell
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a → [a]
```

The other method in is

```haskell
shrink :: (Arbitrary a) ⇒ a → [a]
```

- Maps each value to a number of ‘structurally smaller’ values.
- When a failing test case is discovered, is applied repeatedly until no smaller failing test case can be obtained.

Program coverage

To assess the quality of your test suite, it can be very useful to use GHC's *program coverage* tool:

```
$ ghc -fhpc Suite.hs --make
$ ./Suite
$ hpc report Suite --exclude=Main --exclude=QC
   18% expressions used (30/158)
    0% boolean coverage (0/3)
         0% guards (0/3), 3 unevaluated
       100% 'if' conditions (0/0)
       100% qualifiers (0/0)
        ...
```

This also generates a `.html` file showing which code has (not) been executed.

module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module Prettify2	42%	9/21		23%	8/34		18%	30/158	
Program Coverage Total	42%	9/21		23%	8/34		18%	30/158	

Figure 1: screenshot

```haskell
data Doc = Empty
         | Char Char
         | Text String
         | Line
         | Concat Doc Doc
         | Union Doc Doc
         deriving (Show,Eq)

{-- /snippet Doc --}

instance Monoid Doc where
    mempty  = empty
    mappend = (<>)

{-- snippet append --}
empty :: Doc
(<>)  :: Doc -> Doc -> Doc
{-- /snippet append --}

empty = Empty

Empty <> y = y
x <> Empty = x
x <> y = x `Concat` y

char :: Char -> Doc
char c = Char c
```

Figure 2: screenshot

Loose ends

- Haskell can deal with infinite values, and so can QuickCheck. However, properties must not inspect infinitely many values. For instance, we cannot compare two infinite values for equality and still expect tests to terminate. Solution: Only inspect finite parts.
- QuickCheck can generate functional values automatically, but this requires defining an instance of another class `Coarbitrary` – but showing functional values is problematic.
- QuickCheck has facilities for testing properties that involve `IO`, but this is more difficult than testing pure properties.

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.
- Use it!

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

Further reading

Required:

- Chapter 11 of Real World Haskell
- *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, Koen Claessen and John Hughes

Additional reading:

- *Software Testing with QuickCheck*, John Hughes
- *Smallcheck and lazy smallcheck: automatic exhaustive testing for small values*, Colin Runciman, Matthew Naylor, Fredrik Lindblad
- *Hedgehog*, Jacob Stanley