AFP lecture 6: Correctness and testing

Patrik Jansson, FP group, Chalmers and U. of Gothenburg
(slides by Andres Löh, Well-Typed, formerly Utrecht U.)

January 9, 2015

- ▶ 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?

Universiteit Utrecht

- 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 with Haskell programs
► QuickCheck, an automated testing library/tool for Haskell

Universiteit Utrecht

- ► Understand how to prove simple properties using equational reasoning.
- ► Understand how to define QuickCheck properties and how to use QuickCheck.
- ► Understand how QuickCheck works and how to make QuickCheck usable for your own larger programs.

3.1 Equational reasoning

Referential transparency

3.1

- "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.

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

3-7

SML is (like most languages) not referentially transparent:

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

The expression evaluates to 4.

SML is (like most languages) not referentially transparent:

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

The expression evaluates to 4. The value of f 1 is 1. But

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

evaluates to 3.

Also

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

cannot be replaced by

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

Referential transparency in Haskell

.1

- ▶ Haskell is referentially transparent.
- ▶ The SML example breaks down because Haskell has no untracked side-effects.

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

The type of f is Int → IO Int, not Int → Int as in SML.

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

3-10

- Because of referential transparency, the definitions of functions give us rules for reasoning about Haskell programs.

- Properties regarding datatypes can be proved using induction:

  data $[a] = [] \mid a : [a]$

  To prove $\forall(xs :: [a]).P\ xs$, we prove
  - $P\ []$
  - $\forall(x :: a)\ (xs :: [a]).P\ xs \to P\ (x : xs)$

```
length :: [a] → Int
length []      = 0
length (x : xs) = 1 + length xs

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
```

length :: [a] → Int
length [] = 0
length (x : xs) = 1 + length xs

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

Theorem (Sorting preserves length)

∀(xs :: [a]).length (isort xs) ≡ length xs

length :: [a] → Int
length [] = 0
length (x : xs) = 1 + length xs

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

Theorem (Sorting preserves length)

∀(xs :: [a]).length (isort xs) ≡ length xs

Lemma

∀(x :: a) (ys :: [a]).length (insert x ys) ≡ 1 + length ys

Lemma

$\forall(x :: a)\ (ys :: [a]).\text{length (insert } x\ ys) \equiv 1 + \text{length } ys$

Proof by induction on the list.

Case [ ]:

$$
\begin{array}{ll}
& \text{length (insert } x\ [\,]) \\
\equiv & \{ \text{ Definition of insert } \} \\
& \text{length } [x] \\
\equiv & \{ \text{ Definition of length } \} \\
& 1 + \text{length } [\,]
\end{array}
$$

Lemma

$\forall (x :: a)\ (ys :: [a]).\text{length}\ (\text{insert}\ x\ ys) \equiv 1 + \text{length ys}$

Case y : ys, case $x \leqslant y$:

$\quad\quad$ length (insert x (y : ys))

$\equiv\quad$ { Definition of insert }

$\quad\quad$ length (x : y : ys)

$\equiv\quad$ { Definition of length }

$\quad\quad 1 +$ length (y : ys)

Universiteit Utrecht

Lemma

$\forall (x :: a)\ (ys :: [a]).length\ (insert\ x\ ys) \equiv 1 + length\ ys$

Case $y : ys$, case $x > y$:

$$
\begin{array}{ll}
& length\ (insert\ x\ (y : ys)) \\
\equiv & \{\ \text{Definition of insert}\ \} \\
& length\ (y : insert\ x\ ys) \\
\equiv & \{\ \text{Definition of length}\ \} \\
& 1 + length\ (insert\ x\ ys) \\
\equiv & \{\ \text{Induction hypothesis}\ \} \\
& 1 + (1 + length\ ys) \\
\equiv & \{\ \text{Definition of length}\ \} \\
& 1 + length\ (y : ys)
\end{array}
$$

Universiteit Utrecht

Theorem

$\forall(xs :: [a]).\text{length (isort } xs) \equiv \text{length } xs$

Proof by induction on the list.

Case []:

$$
\begin{array}{ll}
& \text{length (isort [])} \\
\equiv & \{ \text{ Definition of isort } \} \\
& \text{length []}
\end{array}
$$

Universiteit Utrecht

Theorem

$\forall(xs :: [a]).length \ (isort \ xs) \equiv length \ xs$

Case $x : xs$:

$\quad length \ (isort \ (x : xs))$
$\equiv \quad \{ \ Definition \ of \ isort \ \}$
$\quad length \ (insert \ x \ (isort \ xs))$
$\equiv \quad \{ \ Lemma \ \}$
$\quad 1 + length \ (isort \ xs)$
$\equiv \quad \{ \ Induction \ hypothesis \ \}$
$\quad 1 + length \ xs$
$\equiv \quad \{ \ Definition \ of \ length \ \}$
$\quad length \ (x : xs)$

Universiteit Utrecht

- 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 is usually quite lengthy.
- Careful with special cases (laziness):
  - undefined values;
  - infinite values
- It is infeasible to prove properties about every Haskell program using equational reasoning.

- ► Type systems.
- ► Proof assistants.

3.2 QuickCheck

Universiteit Utrecht

- ▶ QuickCheck is a Haskell library developed by Koen Claessen and John Hughes.
- ▶ An embedded domain-specific language (EDSL) for defining properties.
- ▶ Automatic datatype-driven generation of random test data.
- ▶ Extensible by the user.
- ▶ Shrinks failing test cases.

To use QuickCheck in your program:

> **import** Test.QuickCheck

The simplest interface is to use

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

> **class** Testable prop **where**
>   property :: prop $\rightarrow$ Property
> **instance** Testable Bool
> **instance** (Arbitrary a, Show a, Testable prop) $\Rightarrow$
>   Testable (a $\rightarrow$ prop)

▶ Classes declare predicates on types.

> **class** Testable prop **where**
>   property :: prop → Property

Here, any type can either be Testable or not.

▶ If a predicate holds for a type, this implies that the class methods are supported by the type.
For any type prop such that Testable prop, there is a method property :: prop → Property.
Outside of a class declaration, Haskell denotes this type as

> property :: Testable prop ⇒ prop → Property

- ▶ Instances declare which types belong to a predicate.

  instance Testable Bool
  instance (Arbitrary a, Show a, Testable prop) $\Rightarrow$
             Testable (a $\rightarrow$ prop)

  Booleans are in Testable.
  Functions a $\rightarrow$ prop are in Testable if prop is Testable and
  a is in Arbitrary and in Show.

- ▶ Instance declarations have to provide implementations of
  the class methods (in this case, of property), as a proof
  that the predicate does indeed hold for the type.

- ▶ Other functions that use class methods inherit the class
  constraints:

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

instance Testable Bool

sortAscending :: Bool
sortAscending = sort [2, 1] == [1, 2]

sortDescending :: Bool
sortDescending = sort [2, 1] == [2, 1]

Running QuickCheck:

Main⟩ quickCheck sortAscending
+++ OK, passed 100 tests.

Main⟩ quickCheck sortDescending
∗∗∗ Failed! Falsifiable (after 1 test):

- ▶ Nullary properties are static properties.
- ▶ QuickCheck can be used for unit testing.
- ▶ By default, QuickCheck tests 100 times (which is wasteful for static properties, but configurable).

> **instance** (Arbitrary a, Show a, Testable prop) $\Rightarrow$
>   Testable (a $\rightarrow$ prop)

> sortPreservesLength :: ([Int] $\rightarrow$ [Int]) $\rightarrow$ [Int] $\rightarrow$ Bool
> sortPreservesLength isort xs = length (isort xs) == length xs

> Main⟩ quickCheck (sortPreservesLength isort)
> +++ OK, passed 100 tests.

Read parameterized properties as universally quantified.
QuickCheck automatically generates lists of integers.

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

```
import Data.Set
setSort = toList ∘ fromList
```

Universiteit Utrecht

import Data.Set
setSort = toList ∘ fromList

Main⟩ quickCheck (sortPreservesLength setSort)
*** Failed! Falsifiable (after 6 tests and 2 shrinks):
[1, 1]

import Data.Set
setSort = toList ∘ fromList

Main⟩ quickCheck (sortPreservesLength setSort)
*** Failed! Falsifiable (after 6 tests and 2 shrinks):
$[1, 1]$

▶ The function setSort eliminates duplicate elements, therefore a list with duplicate elements causes the test to fail.

▶ QuickCheck shows evidence of the failure, and tries to present minimal test cases that fail (shrinking).

Property 1

A sorted list should be ordered:

sortOrders :: [Int] $\rightarrow$ Bool
sortOrders xs = ordered (sort xs)

ordered :: Ord a $\Rightarrow$ [a] $\rightarrow$ Bool
ordered []   = True
ordered [x]   = True
ordered (x : y : ys) = x $\leqslant$ y $\wedge$ ordered (y : ys)

Property 2

A sorted list should have the same elements as the original list:

sortPreservesElements :: [Int] → Bool
sortPreservesElements xs = sameElements xs (sort xs)

sameElements :: Eq a ⇒ [a] → [a] → Bool
sameElements xs ys = null (xs \\ ys) ∧ null (ys \\ xs)

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

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

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

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

```
Main⟩ let p = sortPreservesLength isort
Main⟩ quickCheck (λxs → collect (null xs) (p xs))
+++ OK, passed 100 tests:
92% False
 8% True
```

Main⟩ quickCheck (λxs → collect (length xs 'div' 10) (p xs))
+++ OK, passed 100 tests:
31% 0
24% 1
16% 2
 9% 4
 9% 3
 4% 8
 4% 6
 2% 5
 1% 7

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

```
Main⟩ quickCheck (λxs → collect xs (p xs))
+++ OK, passed 100 tests:
6% []
1% [9, 4, −6, 7]
1% [9, −1, 0, −22, 25, 32, 32, 0, 9, . . .
. . .
```

Question

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

The function insert preserves an ordered list:

```
implies :: Bool → Bool → Bool
implies x y = ¬ x ∨ y
```

Problematic:

```
insertPreservesOrdered :: Int → [Int] → Bool
insertPreservesOrdered x xs =
    ordered xs 'implies' ordered (insert x xs)
```

Main⟩ quickCheck insertPreservesOrdered
+++ OK, passed 100 tests.

Main⟩ quickCheck insertPreservesOrdered
+++ OK, passed 100 tests.

But:

Main⟩ **let** iPO = insertPreservesOrdered
Main⟩ quickCheck (λx xs $\rightarrow$ collect (ordered xs) (iPO x xs))
+++ OK, passed 100 tests.
88% False
12% True

Only 12 lists have really been tested!

The solution is to use the QuickCheck implication operator:

$(\Longrightarrow) :: (\text{Testable prop}) \Rightarrow \text{Bool} \rightarrow \text{prop} \rightarrow \text{Property}$
instance Testable Property

The type Property allows to encode not only True or False, but also to reject the test case.

$\text{iPO} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Property}$
$\text{iPO x xs} = \text{ordered xs} \Longrightarrow \text{ordered (insert x xs)}$

Now we get:

Main⟩ quickCheck (λx xs → collect (ordered xs) (iPO x xs))
∗∗∗ Gave up! Passed only 43 tests (100% True).

Configuring QuickCheck

```
data Args = Args {
  replay     :: Maybe (StdGen, Int)
  maxSuccess :: Int
  maxDiscard :: Int
  maxSize    :: Int
  }
stdArgs :: Args
stdArgs = Args {replay     = Nothing,
                maxSuccess = 100,
                maxDiscard = 500,
                maxSize    = 100}
quickCheckWith :: Testable prop => Args -> prop -> IO ()
```

Increasing the number of discarded tests may help.
Better solution: use a custom generator (discussed next).

► Generators belong to an abstract data type Gen. Think of Gen 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. We can define default generators for new datatypes by defining instances of class Arbitrary:

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

```
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
```

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]

- ▶ A simple possibility:

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

- ▶ Better:

  instance Arbitrary Int **where**
      arbitrary = sized $(\lambda n \rightarrow \text{choose } (-n, n))$

- ▶ QuickCheck automatically increases the size gradually, up to the configured maximum value.

A bad approach to generating more complex values is a frequency table:

```
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)]
```

Here:

```
liftM  :: (a → b)       → Gen a → Gen b
liftM2 :: (a → b → c) → Gen a → Gen b → Gen c
```

A bad approach to generating more complex values is a frequency table:

```
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)]
```

Here:

```
liftM  :: (a → b)     → Gen a → Gen b
liftM2 :: (a → b → c) → Gen a → Gen b → Gen c
```

Termination is unlikely!

```
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)
```

Why a non-zero probability for Leaf in the second case of
arbitraryTree?

The other method in Arbitrary is

$$\text{shrink} :: (\text{Arbitrary } a) \Rightarrow a \rightarrow [a]$$

- ▶ Maps each value to a number of structurally smaller values.
- ▶ Default definition returns [] and is always safe.
- ▶ When a failing test case is discovered, shrink is applied repeatedly until no smaller failing test case can be obtained.

- ► Both arbitrary and shrink are examples of datatype-generic functions – they can be defined for (almost) any Haskell datatype in a systematic way.
- ► Haskell does not provide any way to write down such an algorithm.
- ► Many extensions and tools do (cf. course on Generic Programming in block 4).

All lists are ordered?

> Main⟩ quickCheck ordered
> +++ OK, passed 100 tests.

Universiteit Utrecht

All lists are ordered?

Main〉 quickCheck ordered
+++ OK, passed 100 tests.

Use type signatures in GHCi to make sure a sensible type is used!

Main〉 quickCheck (ordered :: [Int] → Bool)
∗∗∗ Failed! Falsifiable (after 3 tests and 2 shrinks):
$[0, -1]$

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.

Universiteit Utrecht

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. Also, showing functional values is problematic.

- ▶ 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. Also, showing functional values is problematic.

- ▶ QuickCheck has facilities for testing properties that involve IO, but this is more difficult than testing pure properties.

4.1 Haskell Program Coverage

Program code can be classified:

- unreachable code: code that simply is not used by the program, usually library code
- reachable code: code that can in principle be executed by the program

Program code can be classified:

- ▶ unreachable code: code that simply is not used by the program, usually library code
- ▶ reachable code: code that can in principle be executed by the program

Reachable code can be classified further:

- ▶ covered code: code that is actually executed during a number of program executions (for instance, tests)
- ▶ uncovered code: code that is not executed during testing

Universiteit Utrecht

Reachable uncovered code

Program code can be classified:

- ▶ unreachable code: code that simply is not used by the program, usually library code
- ▶ reachable code: code that can in principle be executed by the program

Reachable code can be classified further:

- ▶ covered code: code that is actually executed during a number of program executions (for instance, tests)
- ▶ uncovered code: code that is not executed during testing

Uncovered code is untested code – it could be executed, and it could do anything!

Universiteit Utrecht

- ▶ HPC (Haskell Program Coverage) is a tool – integrated into GHC – that can identify uncovered code.
- ▶ Using HPC is extremely simple:
  - ▸ Compile your program with the flag `-fhpc`.
  - ▸ Run your program, possibly multiple times.
  - ▸ Run `hpc report` for a short coverage summary.
  - ▸ Run `hpc markup` to generate an annotated HTML version of your source code.

- ▶ HPC can present your program source code in a color-coded fashion.
- ▶ Yellow code is uncovered code.
- ▶ Uncovered code is discovered down to the level of subexpressions! (Most tools for imperative language only give you line-based coverage analyis.)
- ▶ HPC also analyzes boolean expressions:
  - ▶ Boolean expressions that have always been True are displayed in green.
  - ▶ Boolean expressions that have always been False are displayed in red.

QuickCheck and HPC interact well!

- ▶ Use HPC to discover code that is not covered by your tests.
- ▶ Define new test properties such that more code is covered.
- ▶ Reaching 100% can be really difficult (why?), but strive for as much coverage as you can get.