

# Testing

## Haskell and Cryptocurrencies

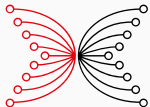
---

Dr. Andres Löb, Well-Typed LLP

Dr. Lars Brünjes, IOHK

Alejandro Garcia, IOHK

2020-08-03



INPUT | OUTPUT

# Goals

- Testing vs Types
- QuickCheck, property-based testing
- Case study for class-based overloading
- Generators
- Haskell program coverage

# What is testing?

---

# What is testing about?

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

# What is testing about?

- 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.*

# What is testing about?

- 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.*
- Exception: Exhaustive testing.

- When is a program correct?

- 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?



# Testing versus types

Do types free us from the need to test?

# Testing versus types

Do types free us from the need to test?

In general, no.

- There are limits to what the type system can express.
- While Haskell's type system is quite expressive, expressing advanced invariants of programs in the types is often a lot of work.
- There's usually a natural balance between compile-time type checking and run-time properties that should be tested.
- However, the presence of types means that we can concentrate on testing the interesting properties.

# Testing in Haskell

There are a quite a number of tools and libraries.

Some noteworthy examples:

- *HUnit* – a classic unit testing library;
- *QuickCheck* – type-driven testing with random test case generation;
- *Hedgehog* – modern testing system in the spirit of QuickCheck;
- *smallcheck* – a variant of QuickCheck supporting exhaustive testing of “small” test cases;
- *test-framework* – integrating different testing libraries into a common framework and Cabal packages;
- *hspec* and *doctest* – integrating testing with documentation.

# Making a choice

We obviously don't have time to look at them all, so we focus on one rather remarkable library:

## QuickCheck

Type-driven testing with random test case generation.

# Making a choice

We obviously don't have time to look at them all, so we focus on one rather remarkable library:

## QuickCheck

Type-driven testing with random test case generation.

QuickCheck shares with other Haskell testing libraries the feature that tests are themselves Haskell programs, and as such they are type checked.

# History of QuickCheck

- 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.

## Example: specifying and testing sorting

---

## A first version of the code

An attempt at insertion sort in 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
```

(This is an example – if you spot errors immediately, ignore them for now ...)



# How to specify sorting?

A good specification is

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

# How to specify sorting?

A good specification is

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

If we want to specify sorting, we should give a specification that distinguishes sorting from all other operations, but does not force 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 xs == length (sort xs)
```

## A first approximation

Certainly, sorting a list should not change its length.

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

We can test by invoking the function `quickCheck`:

```
GHCi> quickCheck sortPreservesLength
*** Failed! Falsifiable (after 4 tests and 2 shrinks):
[0, 0]
```

## 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
```

## 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 : y : ys
                  | otherwise = y : insert x ys
```

## A new attempt

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

Looks better. But have we tested enough?

## Properties are first-class objects

Note that we can define our own testing-inspired abstractions:

```
(f `preserves` p) x = p x == p (f x)  
sortPreservesLength = sort `preserves` length
```

Is sorting the only list function preserving length?



# Properties are first-class objects

Note that we can define our own testing-inspired abstractions:

```
(f `preserves` p) x = p x == p (f x)  
sortPreservesLength = sort `preserves` length
```

Is sorting the only list function preserving length?

```
idPreservesLength = id `preserves` length
```

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

Clearly, the identity function does not sort the list.

## When is a list sorted?

```
sorted :: [Int] -> Bool
sorted []      = True
sorted (x : xs) = ...
```

## When is a list sorted?

```
sorted :: [Int] -> Bool
sorted []           = True
sorted (x : xs) = ...
```

## When is a list sorted?

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

## When is a list sorted?

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

## When is a list sorted?

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

## Testing again

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

## Testing again

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

Or:

```
(f `ensures` p) x = p (f x)  
sortEnsuresSorted = sort `ensures` sorted
```



## Testing again

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

Or:

```
(f `ensures` p) x = p (f x)  
sortEnsuresSorted = sort `ensures` sorted
```

```
GHCi> quickCheck sortEnsuresSorted  
*** Failed! Falsifiable (after 4 tests and 1 shrink):  
[1, 1]  
GHCi> sort [1, 1]  
[1, 1]
```

But this is correct. So what went wrong?

## Specifications can have bugs, too!

```
GHCi> sorted [2, 2, 4]  
False
```

## Specifications can have bugs, too!

```
GHCi> sorted [2, 2, 4]  
False
```

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

## Specifications can have bugs, too!

```
GHCi> sorted [2, 2, 4]  
False
```

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

## Another attempt

```
GHCi> quickCheck sortEnsuresSorted  
*** Failed! Falsifiable (after 5 tests and 4 shrinks):  
[0, 0, - 1]
```

There still seems to be a bug.

## Another attempt

```
GHCi> quickCheck sortEnsuresSorted
*** Failed! Falsifiable (after 5 tests and 4 shrinks):
[0, 0, - 1]
```

There still seems to be a bug.

```
GHCi> sort [0, 0, - 1]
[0, 0, - 1]
```

## Correcting again

```
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
```

## Correcting again

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



## Correcting again

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

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

```
GHCi> quickCheck sortEnsuresSorted
```

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

# 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

```
evilNoSort :: [Int] -> [Int]  
evilNoSort xs = replicate (length xs) 0
```

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

```
f `permutes` xs = f xs `elem` permutations xs  
sortPermutates xs = sort `permutes` xs
```

Our sorting function fulfills this specification, but `evilNoSort` does not.

## How QuickCheck works

---

# How to use QuickCheck

To use QuickCheck in your program:

```
import Test.QuickCheck  -- from package QuickCheck
```

# How to use QuickCheck

To use QuickCheck in your program:

```
import Test.QuickCheck  -- from package QuickCheck
```

Define properties.

# How to use QuickCheck

To use QuickCheck in your program:

```
import Test.QuickCheck  -- from package QuickCheck
```

Define properties.

Then call `quickCheck` to test the properties.

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



## The type of `quickCheck`

The type of `quickCheck` is an *overloaded* type:

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

## The type of `quickCheck`

The type of `quickCheck` is an *overloaded* type:

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

- The argument of `quickCheck` is a property of type `prop`.

## The type of `quickCheck`

The type of `quickCheck` is an *overloaded* type:

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

- The argument of `quickCheck` is a property of type `prop`.
- The only restriction on the type `prop` is that it is in the `Testable` type class.

## The type of `quickCheck`

The type of `quickCheck` is an *overloaded* type:

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

- The argument of `quickCheck` is a property of type `prop`.
- The only restriction on the type `prop` is that it is in the `Testable` type class.
- When executed, `quickCheck` 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 `[Int] -> Bool`:

```
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 **True** for 100 cases, this success is reported in a message.
- If the result is **False** for a case, the test case triggering the result is printed.

## Other forms of properties

```
appendLength :: [a] -> [a] -> 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.)

```
GHCi> quickCheck takeDrop
```

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

```
GHCi> quickCheck dropTwice
```

```
*** Failed! Falsifiable (after 2 tests and 1 shrink):
```

```
1
```

```
-1
```

```
[0]
```

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

```
[0]
```

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

```
[]
```

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

```
[0]
```

## Other forms of properties (contd.)

```
GHCi> quickCheck takeDrop
```

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

```
GHCi> quickCheck dropTwice
```

```
*** Failed! Falsifiable (after 2 tests and 1 shrink):
```

```
1  
-1  
[0]
```

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

```
[0]
```

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

```
[]
```

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

```
[0]
```



# Nullary properties

A property without arguments is also possible:

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

wrong :: Bool
wrong = False

GHCi> quickCheck lengthEmpty
+++ OK, passed 100 tests.

GHCi> quickCheck wrong
*** Failed! Falsifiable (after 1 test):
```

# Nullary properties

A property without arguments is also possible:

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

wrong :: Bool
wrong = False

GHCi> quickCheck lengthEmpty
+++ OK, passed 100 tests.

GHCi> quickCheck wrong
*** Failed! Falsifiable (after 1 test):
```

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 in `Testable`:

- testable properties usually are functions (with arbitrarily many arguments) resulting in a `Bool`

# Properties

Recall the type of `quickCheck`:

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

We can now say more about when types are in `Testable`:

- testable properties usually are functions (with arbitrarily many arguments) resulting in a `Bool`

Are arbitrary argument types admissible?

# Properties

Recall the type of `quickCheck`:

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

We can now say more about when types are in `Testable`:

- testable properties usually are functions (with arbitrarily many arguments) resulting in a `Bool`

Are arbitrary argument types admissible?

No – QuickCheck has to know how to produce random test cases of such types.

## Properties (contd.)

We can express the idea in Haskell using type classes:

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

## Properties (contd.)

We can express the idea in Haskell using type classes:

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

A `Bool` is testable:

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

## Properties (contd.)

We can express the idea in Haskell using type classes:

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

A `Bool` is testable:

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

If a type is testable, we can add another 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 ...
```



## Analyzing the test data

---

# Obtaining information about the test data

## Question

Why is it important to know what data we actually test on?

## Obtaining information about the test data

### Question

Why is it important to know what data we actually test on?

A simple way is to use

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

rather than

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

## Observations about QuickCheck test data

- First test cases seem to be rather small.
- Test cases seem to increase in size over time.
- Duplicate test cases occur.

## Observations about QuickCheck test data

- First test cases seem to be rather small.
- Test cases seem to increase in size over time.
- Duplicate test cases occur.

Often, `verboseCheck` is too much. We want to get information on the distribution of test cases according to a certain property.

## The function `collect`

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

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

## The function `collect`

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

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

```
GHCi> let sPL = sortPreservesLength  
GHCi> quickCheck  
      (\ xs -> collect (null xs) (sPL xs))  
+++ OK, passed 100 tests:  
97% False .  
3% True .
```

## The function `collect` (contd.)

```
GHCi> quickCheck
      (\ xs -> collect (length xs `div` 10)
                      (sPL xs))
```

+++ OK, passed 100 tests:

29% 0

23% 1

14% 2

11% 3

7% 4

6% 5

4% 9

4% 6

2% 7



## The type `Property`

Recall the type of `collect`:

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

The type `Property` is QuickCheck-specific. It holds more structural information about a property than a plain `Bool` ever could.

```
instance Testable Property where ...
```

Like `Bool`, a `Property` is testable, so for us, not much changes.

## Conditions in properties

---

# Implications

The function `insert` preserves an ordered list.

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

A problematic property

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

Can you imagine why?

## Implications (contd.)

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

## Implications (contd.)

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

But:

```
GHCi let iPO = insertPreservesOrdered  
GHCi 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 for the result.

## Implications (contd.)

The solution is to use the QuickCheck implication operator:

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

We see `Property` again – this type allows us to encode not only `True` or `False`, but also to reject 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:

```
GHCi quickCheck
      (\ x xs -> collect (sorted xs) (iP0 x xs))
*** Gave up! Passed only 41 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.

## Custom generators

---



# Generators

- Generators belong to an abstract data type `Gen`.
- 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  
  ...
```

# Generators

- Generators belong to an abstract data type `Gen`.
- 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  
  ...
```

Think of a `Gen a` as an abstract set of information on how to produce values of type `a` randomly.

# Running generators

QuickCheck has internal functions to extract random values from generators.

For end users, two debugging functions are offered:

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

These produce a number of random values generated by the given `Gen a`, and print them in the case of `sample`, or return them in the case of `sample'`.

## Building new generators

QuickCheck includes a library for the construction of new generators:

```
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

For enumeration types, defining generators is easy:

```
instance Arbitrary Bool where
  arbitrary = elements [False, True]
instance Arbitrary Dir where
  arbitrary = elements [North, East, South, West]
```

## Generating numbers

A simple possibility:

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

Better:

```
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:

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

## 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:

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

### Example

```
GHCi> mkSorted [1, 3, - 4, 0, 2]
[1, 4, 8, 8, 10]
```



## How to generate sorted lists (contd.)

The original generator can be adapted as follows:

```
genSorted :: Gen [Int]  
genSorted = fmap mkSorted arbitrary
```

Yes, `Gen` is an instance of `Functor`!

## Using a custom generator

There is another function to construct properties:

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

```
iPO :: Int -> Property  
iPO x = forall genSorted  
  (\ xs -> sorted xs ==> sorted (insert x xs))
```

## Using a custom generator

There is another function to construct properties:

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

```
iP0 :: Int -> Property  
iP0 x = forAll genSorted  
  (\ xs -> sorted xs ==> sorted (insert x xs))
```

```
GHCi> quickCheck iP0  
+++ OK, passed 100 tests.
```

# Modifiers

The module `Test.QuickCheck.Modifiers` defines a number of `newtype` wrappers:

```
newtype Positive a      = Positive a
newtype NonNegative a   = NonNegative a
newtype NonZero a       = NonZero a
newtype NonEmptyList a = NonEmpty [a]
newtype OrderedList a   = Ordered [a]
```

These types have different `Arbitrary` instances than their underlying types, implementing a number of frequently required additional invariants.

## Using modifiers

So, instead of hand-writing our own generator for sorted lists, we could have used:

```
iPO :: Int -> OrderedList Int -> Bool  
iPO x (Ordered xs) = sorted (insert x xs)
```

## Using modifiers

So, instead of hand-writing our own generator for sorted lists, we could have used:

```
iPO :: Int -> OrderedList Int -> Bool  
iPO x (Ordered xs) = sorted (insert x xs)
```

The **newtype** wrapper technique for non-standard class instances is also applicable for your own generators, and also applicable in completely different situations.

All lists are sorted?

```
GHCi> quickCheck sorted
```

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

# GHCi pitfall

All lists are sorted?

```
GHCi> quickCheck sorted
+++ OK, passed 100 tests.
```

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

```
GHCi> quickCheck (sorted :: [Int] -> Bool)
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
[0, - 1]
```



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.
- You should use it.
- The `smallcheck`, `hedgehog` and `HUnit` libraries are also worth checking out.

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

# Haskell Program Coverage

---

# 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 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

# 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!*

# Introducing HPC

- 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.
- It is even easier with cabal:  
`cabal test --enable-coverage`.

# What HPC does

- 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 analysis.)
- 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.