QuickCheck

An introduction

Daniel Larsson

GroupTalk AB

January 31, 2017

What is QuickCheck?

- Library for unit testing your code

What is QuickCheck?

- Library for unit testing your code
- You write *invariants* for your code

What is QuickCheck?

- Library for unit testing your code
- You write *invariants* for your code
- The library generates test data

What is QuickCheck?

- Library for unit testing your code
- You write *invariants* for your code
- The library generates test data

What is QuickCheck?

- Library for unit testing your code
- You write *invariants* for your code
- The library generates test data

$$prop\_reverse\_identity :: Eq\ a \Rightarrow [a] \rightarrow Bool$$
$$prop\_reverse\_identity\ s = s \equiv (reverse \circ reverse)\ s$$

What is QuickCheck?

- Library for unit testing your code
- You write *invariants* for your code
- The library generates test data

$$prop\_reverse\_identity :: Eq\ a \Rightarrow [a] \rightarrow Bool$$
$$prop\_reverse\_identity\ s = s \equiv (reverse \circ reverse)\ s$$

What is QuickCheck?

- Library for unit testing your code
- You write *invariants* for your code
- The library generates test data

$$
\begin{aligned}
&prop\_reverse\_identity :: Eq\ a \Rightarrow [a] \rightarrow Bool \\
&prop\_reverse\_identity\ s = s \equiv (reverse \circ reverse)\ s
\end{aligned}
$$

$$
\begin{aligned}
&prop\_reverse\_ends\ s = compareEnds\ s\ (reverse\ s) \\
&\quad \mathit{where}\ compareEnds\ [\,]\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \_ = True \\
&\qquad\qquad compareEnds\ s@(h : t)\ r = \\
&\qquad\qquad\quad \mathit{let}\ rl = last\ r \\
&\qquad\qquad\qquad\ \ ri = init\ r \\
&\qquad\qquad\quad \mathit{in}\ h \equiv rl \wedge compareEnds\ t\ ri
\end{aligned}
$$

What is QuickCheck?

- Library for unit testing your code
- You write *invariants* for your code
- The library generates test data

$$
\begin{aligned}
&prop\_reverse\_identity :: Eq\ a \Rightarrow [a] \rightarrow Bool \\
&prop\_reverse\_identity\ s = s \equiv (reverse \circ reverse)\ s
\end{aligned}
$$

$$
\begin{aligned}
&prop\_reverse\_ends\ s = compareEnds\ s\ (reverse\ s) \\
&\quad \mathit{where}\ compareEnds\ [\,]\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \_ = True \\
&\qquad\qquad compareEnds\ s@(h : t)\ r = \\
&\qquad\qquad\quad \mathit{let}\ rl = last\ r \\
&\qquad\qquad\qquad\ \ ri = init\ r \\
&\qquad\qquad\quad \mathit{in}\ h \equiv rl \wedge compareEnds\ t\ ri
\end{aligned}
$$

An example, encoding a string

Here is a nonsense encoder, converting a Text into a ByteString.

```
encode :: Text -> BS.ByteString
encode = B.toLazyByteString . prepend terminator <$> enc
  where enc        :: Text -> Builder
        enc        = T.foldr (B.append . encChar) B.empty
        encChar    = B.singleton . fromIntegral . ord
        prepend    = flip B.append
        terminator = encChar '\0'
```

An example, decoding a string

This is the inverse of encode, we decode a ByteString, returning a Text

```
decode :: BS.ByteString → Text
decode = T.unfoldr decodeChar
  where decodeChar :: ByteString → Maybe (Char, ByteString)
        decodeChar bs =
          let ch = BS.head bs
          in if ch ≡ 0
               -- We found the end, stop the unfold
             then Nothing
               -- Return the converted character, and the
               -- remainder of the string we need to convert
             else Just (chr $ fromIntegral ch, BS.tail bs)
```

An example, property

As mentioned, `decode` and `encode` should be inverses of each other. We can express this with the following property:

$$prop\_reversible\ s = s \equiv (decode \circ encode)\ s$$

QuickCheck can help us verify that this property is true:

$$quickCheck\ prop\_reversible$$

An example, property

As mentioned, `decode` and `encode` should be inverses of each other. We can express this with the following property:

$$prop\_reversible\ s = s \equiv (decode \circ encode)\ s$$

QuickCheck can help us verify that this property is true:

$$quickCheck\ prop\_reversible$$

An example, fixing the bug

```
encode2 :: Text → BS.ByteString
encode2 = B.toLazyByteString ∘ prepend terminator ⟨$⟩ enc
  where enc        :: Text → Builder
        enc        = T.foldr (B.append ∘ encChar) B.empty
        prepend    = flip B.append
        terminator = encChar '\0'

charBuilder :: Char → Builder
charBuilder = B.singleton ∘ fromIntegral ∘ ord

encChar       :: Char → Builder
encChar '\0' = charBuilder '\\' `B.append` charBuilder '\0'
encChar '\\' = charBuilder '\\' `B.append` charBuilder '\\'
encChar c    = charBuilder c
```

An example, fixing the bug

```
decode2 :: BS.ByteString → Text
decode2 = T.unfoldr decodeChar
  where decodeChar    :: ByteString → Maybe (Char, ByteString)
        decodeChar bs =
          let ch  = BS.head bs
              chr' = chr ∘ fromIntegral
          in if ch ≡ 0
               -- We found the end, stop the unfold
             then Nothing
             else if chr' ch ≢ '\\'
                    -- Return the converted character, and the
                    -- remainder of the string we need to convert
                  then Just (chr' ch, BS.tail bs)
                    -- Read the next character
                  else let ch' = BS.head $ BS.tail bs
                       in Just (chr' ch', BS.tail $ BS.tail bs)
```

An example, property

Identical property, with the updated encode/decode implementations

$$prop\_reversible2\ s = s \equiv (decode2 \circ encode2)\ s$$

Let us check the property again with QuickCheck

$$quickCheck\ prop\_reversible2$$

An example, property

Identical property, with the updated encode/decode implementations

$$prop\_reversible2\ s = s \equiv (decode2 \circ encode2)\ s$$

Let us check the property again with QuickCheck

$$quickCheck\ prop\_reversible2$$

An example, fixing the bug

```
encode3 :: Text → BS.ByteString
encode3 = B.toLazyByteString ∘ prepend terminator ⟨$⟩ enc
  where enc        :: Text → Builder
        enc        = T.foldr (B.append ∘ encChar) B.empty
        prepend    = flip B.append
          -- This line changed. We can't call encChar here,
          -- since it will escape the character!
        terminator = charBuilder '\0'
```

An example, property

Okay, testing with the 3rd version of encode.

$$prop\_reversible3\ s = s \equiv (decode2 \circ encode3)\ s$$

$$quickCheck\ prop\_reversible3$$

An example, property

Okay, testing with the 3rd version of encode.

$$prop\_reversible3\ s = s \equiv (decode2 \circ encode3)\ s$$

$$quickCheck\ prop\_reversible3$$

Finally!

An example, property

...or?

$$prop\_reversible3\ \texttt{"5.2‰"}$$

An example, property

...or?

$$prop\_reversible3\ \texttt{"5.2‰"}$$

$$False$$

An example, property

...or?

$prop\_reversible3$ "5.2‰"

$False$

Hmmmm... our encoder/decoder is too simplistic, it doesn't handle multibyte unicode characters. And the standard test data generator for the Text datatype doesn't generate multibyte characters either, so this isn't being detected.

Other libraries and tools for testing

- SmallCheck - Similar to QuickCheck, but exhaustive test case generation, rather than randomized

Other libraries and tools for testing

- SmallCheck - Similar to QuickCheck, but exhaustive test case generation, rather than randomized
- HUnit - Your regular xUnit test tool

Other libraries and tools for testing

- SmallCheck - Similar to QuickCheck, but exhaustive test case generation, rather than randomized
- HUnit - Your regular xUnit test tool
- HSPec - Inspired by Ruby's RSpec. Can incorporate QuickCheck tests

Other libraries and tools for testing

- SmallCheck - Similar to QuickCheck, but exhaustive test case generation, rather than randomized
- HUnit - Your regular xUnit test tool
- HSPec - Inspired by Ruby's RSpec. Can incorporate QuickCheck tests
- tasty - A testing framework for organizing tests. The actual test cases can be written using any of the above

Other libraries and tools for testing

- SmallCheck - Similar to QuickCheck, but exhaustive test case generation, rather than randomized
- HUnit - Your regular xUnit test tool
- HSPec - Inspired by Ruby's RSpec. Can incorporate QuickCheck tests
- tasty - A testing framework for organizing tests. The actual test cases can be written using any of the above
- But, use the type system and the compiler to your advantage!

QuickCheck internals

So, Haskell is a (very) strongly typed language, what the heck is the type of `quickCheck`?

QuickCheck internals

So, Haskell is a (very) strongly typed language, what the heck is the type of `quickCheck`?

QuickCheck internals

So, Haskell is a (very) strongly typed language, what the heck is the type of `quickCheck`?

```
class Testable prop where
  property   :: prop -> Property
  exhaustive :: prop -> Bool
  exhaustive _ = False

instance Testable Bool

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

- To be `Testable`, the type needs to be convertable to a `Property`.

QuickCheck internals

So, Haskell is a (very) strongly typed language, what the heck is the type of `quickCheck`?

```
class Testable prop where
  property   :: prop -> Property
  exhaustive :: prop -> Bool
  exhaustive _ = False

instance Testable Bool

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

- To be `Testable`, the type needs to be convertable to a `Property`.
- A function is `Testable` if the result is `Testable`, and the argument is `Arbitrary` and can be converted to a string (`Show`)

QuickCheck internals

```
class Arbitrary a where
    -- A generator for values of the given type.
  arbitrary :: Gen a
  arbitrary = error "no default generator"

    -- Produces a (possibly) empty list of all the possible
    -- immediate shrinks of the given value.
  shrink :: a -> [a]
  shrink _ = []
```

`Gen` is a monad for producing random test data. Instances of `Arbitrary` are responsible for creating random values.

QuickCheck internals

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

instance Arbitrary a ⇒ Arbitrary (Maybe a) where
  arbitrary = frequency [(1, return Nothing)
                        ,(3, liftM Just arbitrary)
                        ]
  shrink (Just x) = Nothing : [Just x' | x' ← shrink x]
  shrink _        = []

instance Arbitrary TS.Text where
  arbitrary = TS.pack ⟨$⟩ arbitrary
  shrink xs = TS.pack ⟨$⟩ shrink (TS.unpack xs)

instance Arbitrary Char where
  arbitrary = chr `fmap` oneof [choose (0, 127), choose (0, 255)]
```