Property Based Testing

WHAT IS A PROPERTY TEST?

WHAT IS A PROPERTY TEST?

A condition holds across a range of data

Invertable

```
reverse (reverse x) === x
```

Associative

```
a + (b + c) === (a + b) + c
```

Commutative

```
a + b === b + a
```

Idempotent

```
logout >> logout === logout
```

Round Tripping

```
parse (print x) === x
```

WHY CARE?

- Cover a larger part of your inputs
- Finds edge cases you didn't think of
- Automatically reduce to a minimal case
- Plays nicely with existing tests

HOW THEY WORK

- Generate some data
- Test the property
- Shrink the data if the property fails
- Test again

GENERATING DATA

Generated not hard coded

Generated not hard coded

Built from composable primitive generators

Generated not hard coded

Built from composable primitive generators

Filtering for desired traits

All my code examples will be written using

Haskell

Hedgehog

Java

jqwik

with some minor handwaving

```haskell
text' :: MonadGen m => m Text
text' = Gen.text
  (Range.linear 1 10)
  (Gen.filter (/= '\0') Gen.unicode)
```

```java
@Provide
Arbitrary<String> text() {
  return Arbitraries.strings().all().excludeChars('\u0000')
    .ofMinLength(1).ofMaxLength(10);
}
```

```haskell
text' :: MonadGen m => m Text
text' = Gen.text
  (Range.linear 1 10)
  (Gen.filter (/= '\0') Gen.unicode)
```

```java
@Provide
Arbitrary<String> text() {
  return Arbitraries.strings().all().excludeChars('\u0000')
    .ofMinLength(1).ofMaxLength(10);
}
```

This is telling the testing library to make a string of 1 to 10 unicode characters, excluding the null character

```haskell
text' :: MonadGen m => m Text
text' = Gen.text
  (Range.linear 1 10)
  (Gen.filter (/= '\0') Gen.unicode)
```

```java
@Provide
Arbitrary<String> text() {
  return Arbitraries.strings().all().excludeChars('\u0000')
    .ofMinLength(1).ofMaxLength(10);
}
```

This is telling the testing library to make a string of 1 to 10 unicode characters, excluding the null character

If your code breaks on funky characters like direction indicators, bells, or backspace this will find it

Generated data is composed to make larger structures

Describing the smallest part of your data

How to compose with bigger generators

```haskell
newtype Password = Password Text
newtype Email = Email Text
data User = User Email Password
genPassword = Password <$> text'
genEmail = Email <$> text'
genUser :: MonadGen m => m User
genUser = User <$> genEmail <*> genPassword
```

```java
@Provide
Arbitrary<Password> genPassword() {
  Arbitrary<String> password = Arbitraries.strings().all()
    .ofMinLength(3).ofMaxLength(21);
  return Combinators.combine(password)
    .as((pass) -> new Password(pass))
}
@Provide
Arbitrary<User> genUser() {
  Arbitrary<Email> email = ...
  Arbitrary<Password> password = ...
  return Combinators.combine(email, password)
    .as((e, p) -> new User(e, p))
}
```

TESTING THE PROPERTY

Now that we can generate our various inputs we need to test our code

Now that we can generate our various inputs we need to test our code

What to test against if we don't know the answer ahead of time?

Known good functions

- Slow but accurate sorts
- Exhaustive searches
- Expensive models

Mathematical properties

- Invertable
- Associative
- Commutative
- Idempotent
- Round Tripping

If you can say a general thing about a function you can test that

```
list = Gen.list (Range.linear 0 100) Gen.unicode

prop_reverse =
  property $ do
    xs <- forAll list
    reverse (reverse xs) === xs
```

```
@Property
boolean listReversal(@ForAll List<int> list) {
  List<int> list2 = new List(list);
  Collections.reverse(list2);
  Collections.reverse(list2);
  return Arrays.equals(
    list.toArray(new int[0]),
    list2.toArray(new int[0])
  );
}
```

Handful of base assertions

- success
- failure
- equal
- not equal

plus anything language specific

Checking data structures

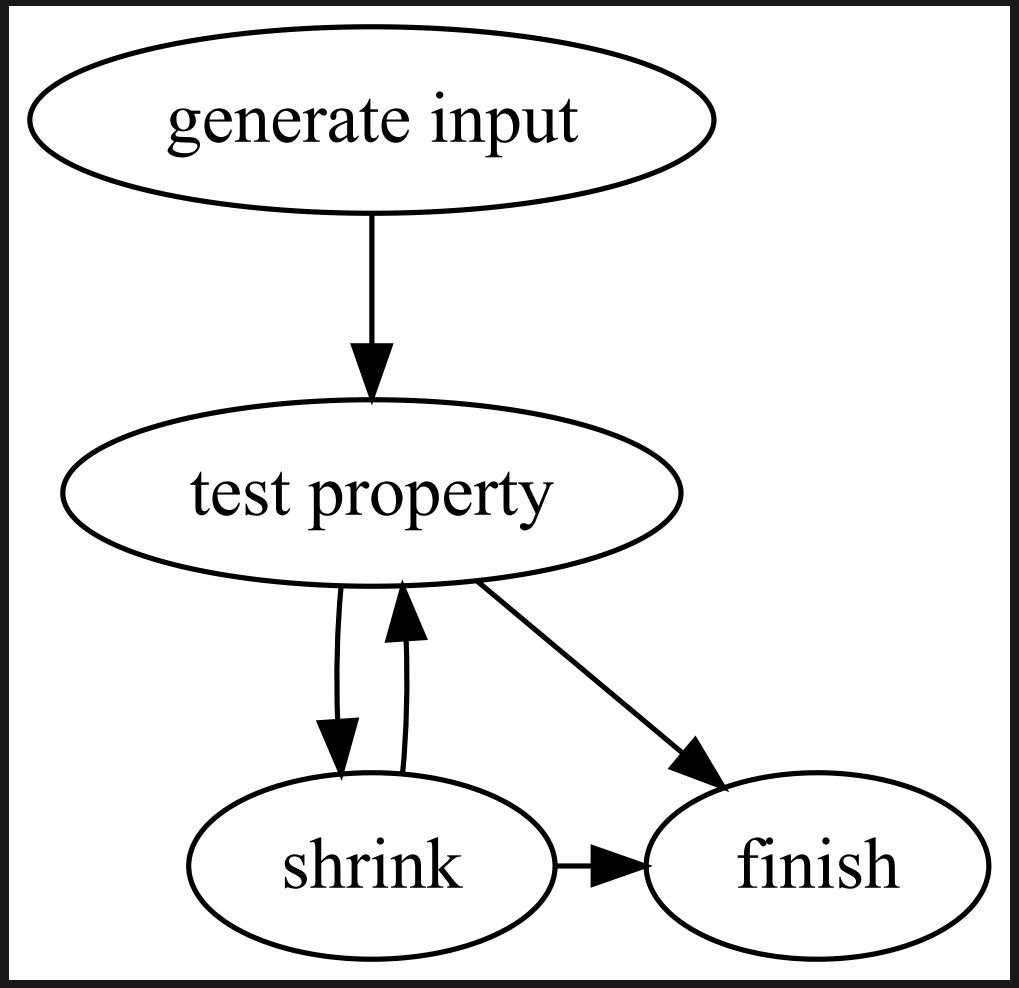
```haskell
isRight (Left _)  = failure
isRight (Right _) = success

prop_alwaysRight = property $ do
  e <- forAll $ Gen.either Gen.unicode Gen.bool
  isRight e
```

```java
// Either<a, b> I'm using is from functionaljava
@Property
boolean alwaysRight(@ForAll Either<a, b> e) {
  return either.isRight(e);
}
```

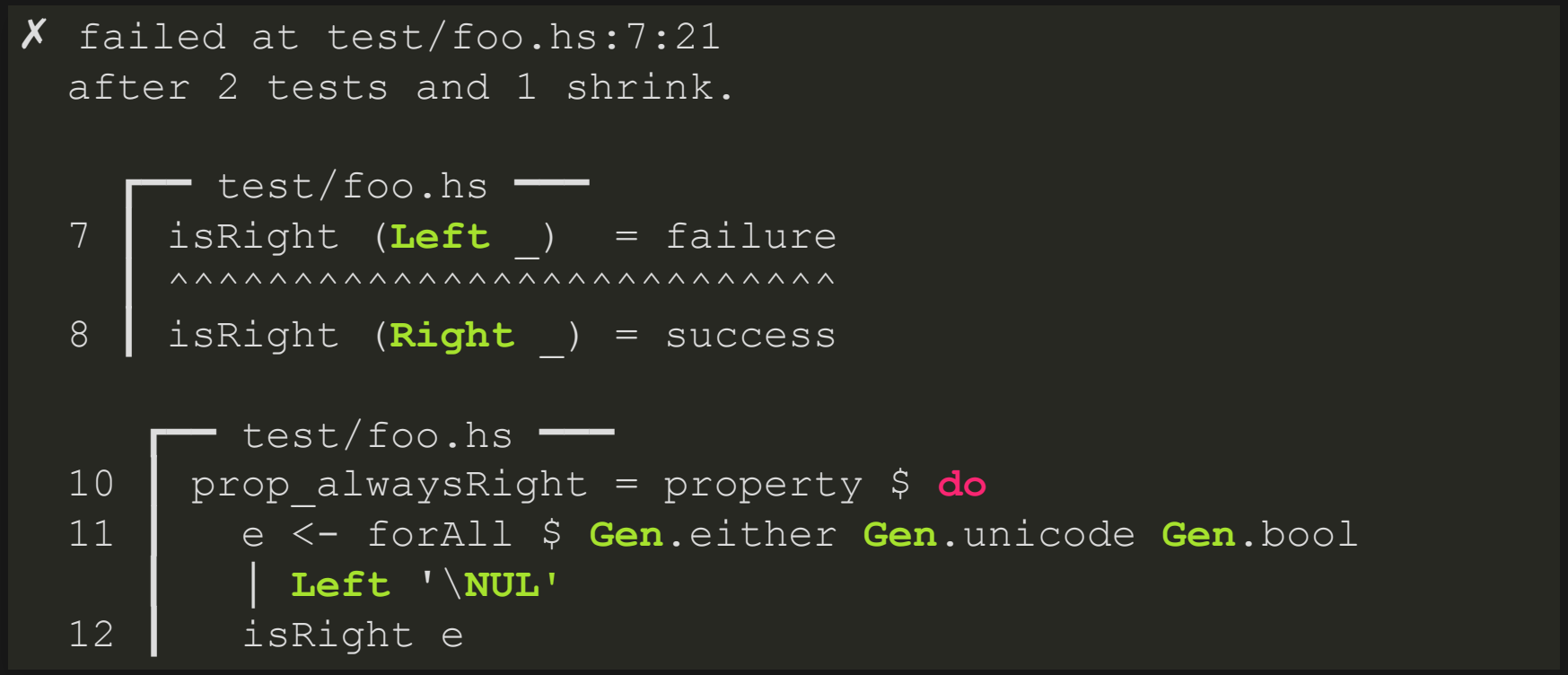
SHRINKING FAILING EXAMPLES

When a failing input is found for a property several things happen

- The generator and seed are saved
- A shrink function is used to make 0 or more smaller values
- These smaller values are recursively tested


generate input
test property
shrink
finish

If a smaller value cannot be generated the testing stops with the last failing value found.


✗ failed at test/foo.hs:7:21
after 2 tests and 1 shrink.
test/foo.hs
7 isRight (Left _) = failure
^^^^^^^^^^^^^^^^^^^^^^^^^^
8 isRight (Right _) = success
test/foo.hs
10 prop_alwaysRight = property $ do
11 e <- forAll $ Gen.either Gen.unicode Gen.bool
| Left '\NUL'
12 isRight e

BUT OWEN MY PROJECT IS ALL STATE-Y!

STATE MACHINES

Stateful applications require a bit more effort

Stateful applications require a bit more effort

Model the state of the target application

Stateful applications require a bit more effort

Model the state of the target application

Updates, deletes, etc

Stateful applications require a bit more effort

Model the state of the target application

Updates, deletes, etc

Run the state changes for app and model

Stateful applications require a bit more effort

Model the state of the target application

Updates, deletes, etc

Run the state changes for app and model

Check that expected and actual state match

Stateful applications require a bit more effort

Model the state of the target application

Updates, deletes, etc

Run the state changes for app and model

Check that expected and actual state match

Excellent blog post on this at the end

Testing Model

```
data TestUser (v :: * -> *) = TestUser
  { userEmail       :: Email
  , userGivenName   :: GivenName
  , userSurname     :: Surname
  ...
  }

newtype AuthState (v :: * -> *) = AuthState
  { authUsers :: Map Email (TestUser v)
  } deriving (Eq, Show)
```

State machine command

```
cNewUser env =
  Command gen (newUserExe env)
    [ Require newUserPrecondition
    , Update newUserUpdate
    , Ensure newUserPostcondition
    ]
  where
    emailNotUsed s e =
      not . any (ilike e) . M.keys $ authUsers s
    gen state = pure $ AuthNewUser
      <$> Gen.filter (emailNotUsed state) genEmail
      <*> fmap GivenName text'
      <*> ...
```

Running a state machine

```
prop_state_machine :: Property
prop_state_machine = property $ do
  actions <- forAll $ Gen.executeSequential
    (Range.linear 1 100)
    initialState
    [cNewUser, ...]
  executeSequential initialState actions

tests :: IO Bool
tests :: checkSequential $$(discover)
```

State machines for jqwik are similar

Taken from https://jqwik.net/docs/current/user-guide.html#stateful-testing

```
class PopAction implements Action<MyStringStack> {
  public boolean precondition(MyStringStack stack) {
    return !stack.isEmpty();
  }
  public MyStringStack run(MyStringStack stack) {
    int sizeBefore = stack.size();
    String topBefore = stack.top();
    String popped = stack.pop();
    Assertions.assertThat(popped).isEqualTo(topBefore);
    Assertions.assertThat(stack.size())
      .isEqualTo(sizeBefore - 1);
    return stack;
  }
}
```

While wonderful what do I use?

LIBRARIES

Haskell

- Hedgehog (also available for F#/C#, R, and Scala)
- Quickcheck

Javascript

- JSVerify

Java

- jqwik

Python

- Hypothesis

FURTHER READING

QuickCheck Manual

Hypothesis Documentation

Introduction to state machine testing: part 1
by Andrew McMiddlin

Property based state machine testing
by Andrew McMiddlin