Property Based Testing

WHO AM I?

Owen Harvey

Software Developer

CSIRO Data61: E&D

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
```

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

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

```
text' = Gen.text
  (Range.constant 1 10)
  (Gen.filter (/= '\0') Gen.unicode)
```

```
text' = Gen.text
  (Range.constant 1 10)
  (Gen.filter (/= '\0') Gen.unicode)
```

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

```
text' = Gen.text
  (Range.constant 1 10)
  (Gen.filter (/= '\0') Gen.unicode)
```

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

You start by describing the smallest part of your data, what it needs, and then use that generator in the next level up

```haskell
genPassword = Password <$> text'

genEmail = Email <$> text'

genUser = User <$> genEmail <*> genPassword
```

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

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

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

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

Handful of base assertions

- success
- failure
- equal
- not equal

plus anything language specific

Checking data structures and types

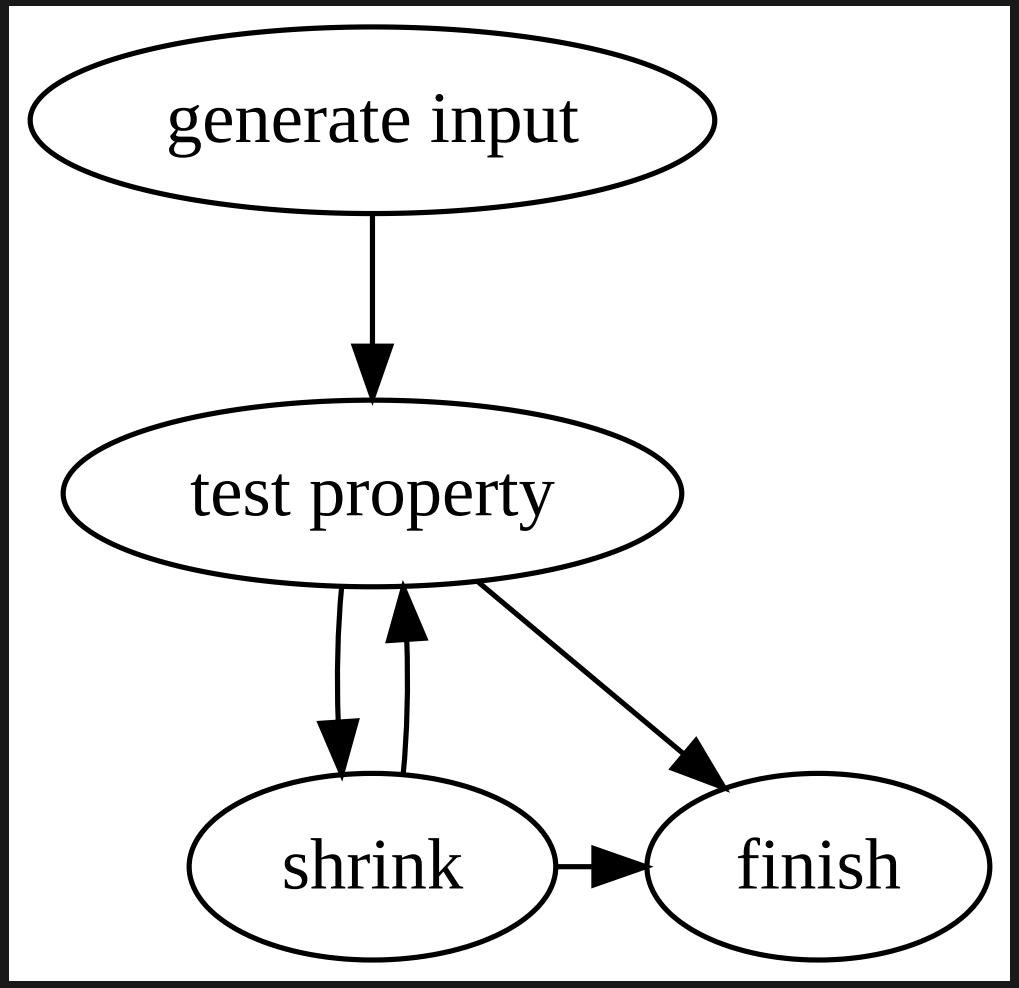
```
isRight (Left _)  = failure
isRight (Right _) = success

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

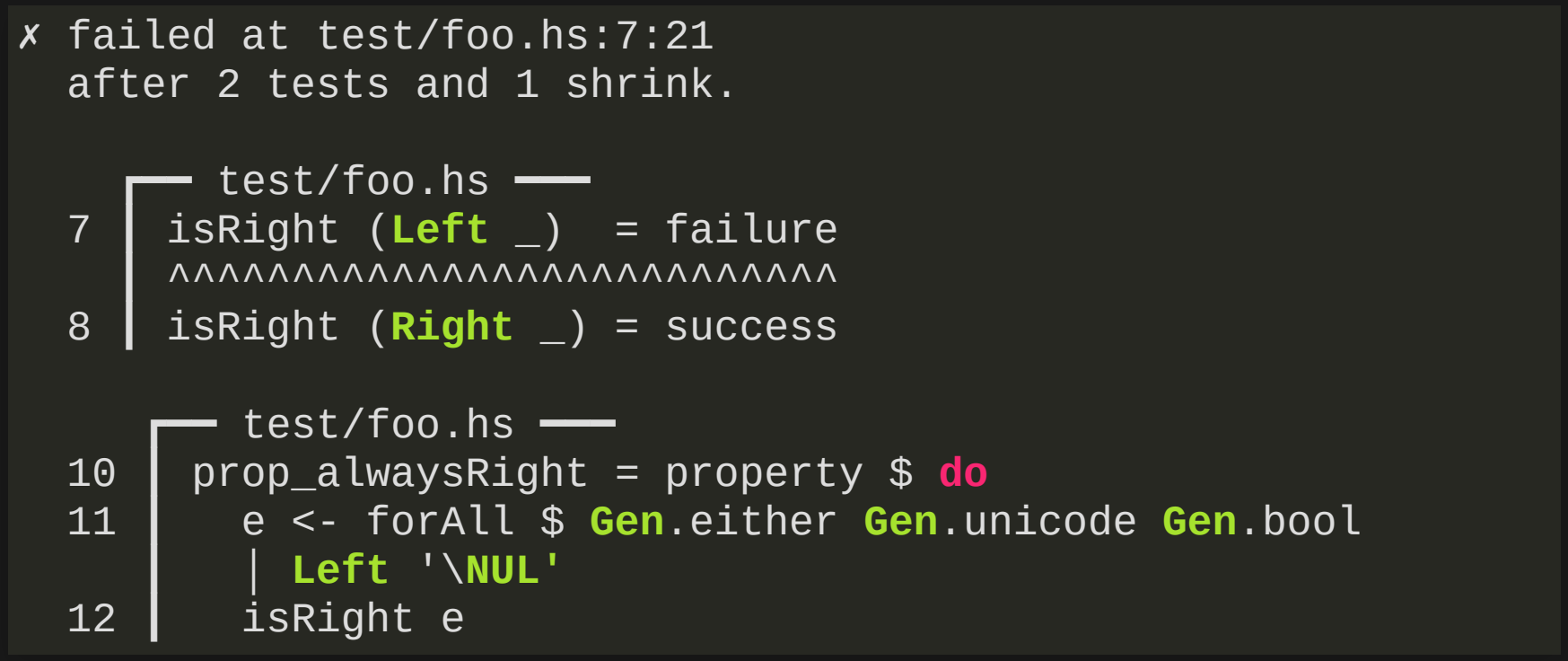
SHRINKING FAILING EXAMPLES

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 requires a bit more effort

Stateful applications requires a bit more effort

Model the state of the target application

Stateful applications requires a bit more effort

Model the state of the target application

Updates, deletes, etc

Stateful applications requires a bit more effort

Model the state of the target application

Updates, deletes, etc

Run the state changes for app and model

Stateful applications requires 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 requires 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

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

While wonderful what do I use?

LIBRARIES

Haskell

Hedgehog

Quickcheck

Javascript

JSVerify

Search "$LANGUAGE quickcheck"

FURTHER READING

QuickCheck Manual

Introduction to state machine testing: part 1
by Andrew McMiddlin

Property based state machine testing
by Andrew McMiddlin