CIS 1904: Haskell

Introduction to Testing

Logistics

- HW 01 due yesterday
- HW 02 will be released tomorrow

What are some important properties of code?

Correctness

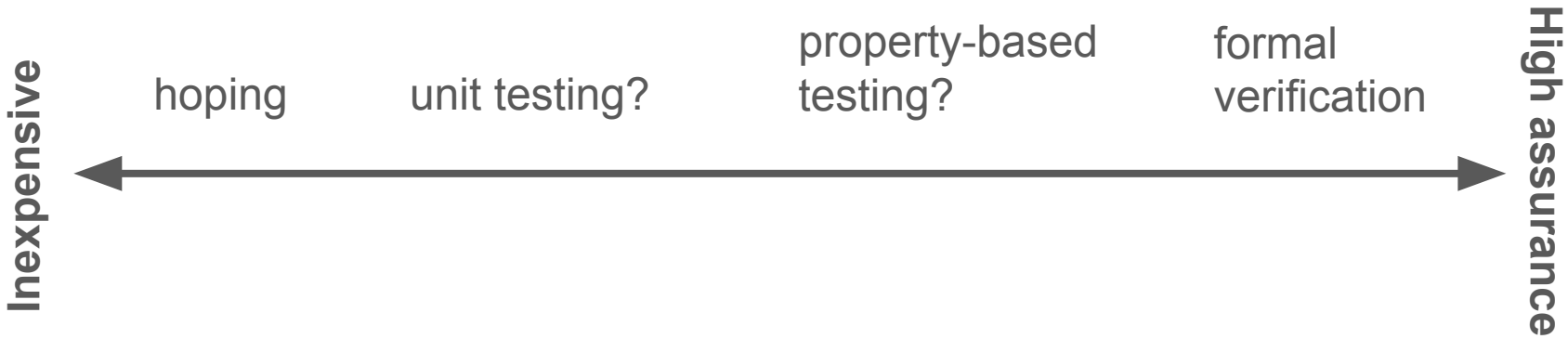
Correctness

How do we write tests?

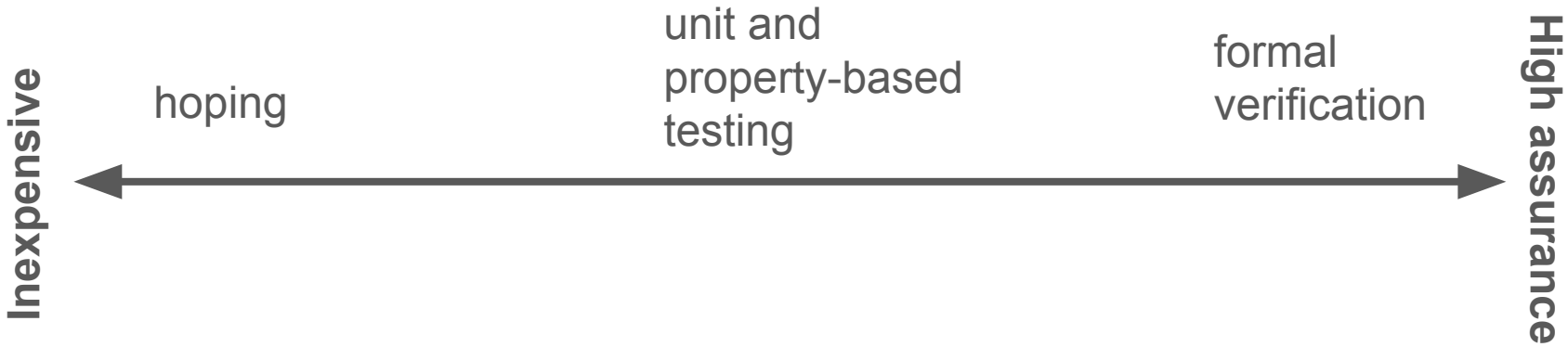
How do we write tests?

- Write a few input/output examples (Unit testing)
- Randomly generate many input/output examples (Property-based testing)

Correctness



Correctness



HUnit

- Haskell’s unit testing library
- Based on JUnit

```
testExample1 :: Test
testExample1 =
  "example1"
    ~: [ "four" ~: example1 4 ~?= 16,
         "zero" ~: example1 0 ~?= 0
       ]
```

```
testExample1 :: Test
testExample1 =
  "example1"
    ~: [ "four" ~: example1 4 ~?= 16,
         "zero" ~: example1 0 ~?= 0
       ]
```

```
testExample1 :: Test
testExample1 =
  "example1"
    ~: [ "four" ~: example1 4 ~?= 16,
         "zero" ~: example1 0 ~?= 0
       ]
```

```
testExample1 :: Test
testExample1 =
  "example1"
    ~: [ "four" ~: example1 4 ~?= 16,
         "zero" ~: example1 0 ~?= 0
       ]
```

```
testExample1 :: Test
testExample1 =
  "example1"
    ~: [ "four" ~: example1 4 ~?= 16,
         "zero" ~: example1 0 ~?= 0
       ]
```

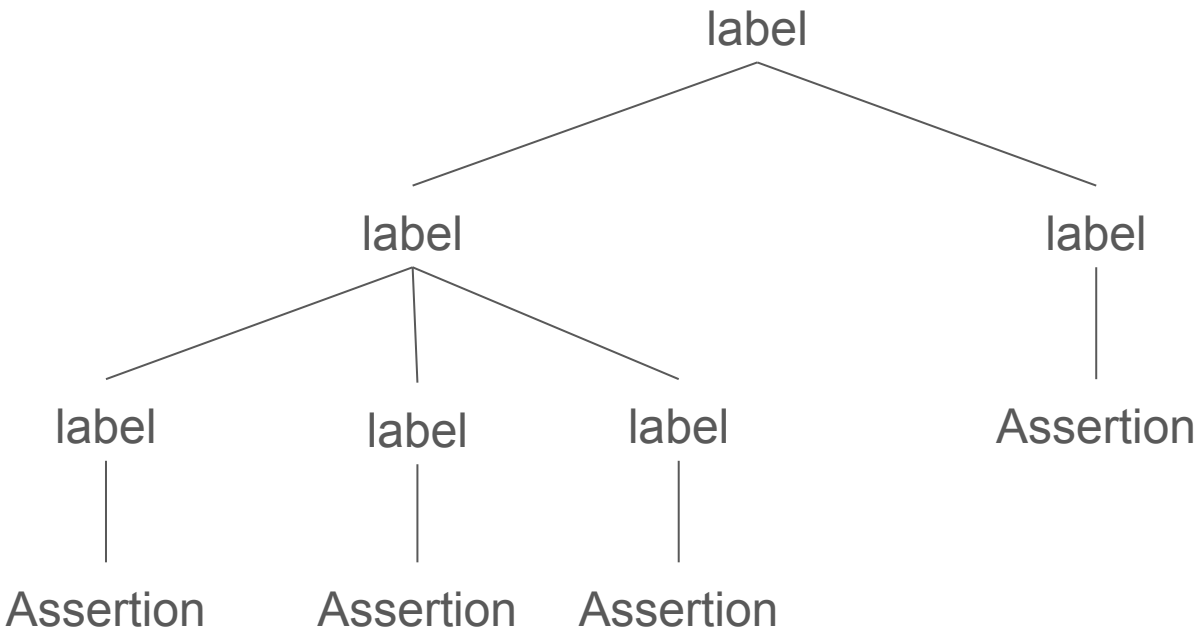
```
stack ghci
```

```
ghci> runTestTT testExample1
```

```
stack ghci
```

```
ghci> runTestTT testExample1
Cases: 2  Tried: 2  Errors: 0  Failures: 0
```

```
testExample1 :: Test
testExample1 =
  "example1"
    ~: [ "four" ~: example1 4 ~?= 16,
         "zero" ~: example1 0 ~?= 0
       ]
```


label
label
label
label
label
label
Assertion
Assertion
Assertion
Assertion

```
testExample1 :: Test
testExample1 =
  "example1"
    ~: [ "four" ~: example1 4 ~?= 16,
         "zero" ~: example1 0 ~?= 0
       ]
```

```
stack ghci
```

```
ghci> runTestTT testExample1
### Failure in: 0:example1:zero
expected: 0
 but got: 7
```

```
take :: Int -> [Int] -> [Int]
take = undefined
```

```
take :: Int -> [Int] -> [Int]
take = Data.List.take

testTake :: Test
testTake =
  "take"
    ~: [ "empty" ~: take 1 [] ~?= [],
         "negative" ~: take (-1) [5,0,8,1] ~?= [],
         “regular” ~: take 2 [5,0,8] ~?= [5,0]
       ]
```

Testing

Strategy so far: Write a few example inputs and expected outputs

Testing

Strategy so far: Write a few example inputs and expected outputs

- Manual effort proportional to number of test cases → few test cases
- Easy to miss entire categories of input (e.g., negative numbers)
  - Easy to make mistakes in expected output

Property-Based Testing

Write tests for *properties* that should hold of all output.

E.g., the output of take should:

- Be a prefix of the input list
- Have length n, for 0 <= n < length xs
- etc.

We can then *randomly generate* arbitrarily many inputs and check that the property holds for all of the outputs.

Libraries

- Hypothesis (Python)
- JUnit-quickcheck, JQF, Jqwik (Java)
- **QuickCheck (Haskell)**
- theft - (C)
- fast-check (JavaScript)
- …

QuickCheck

```
prop_prefix :: [Int] -> Int -> Bool
prop_prefix i xs = isPrefixOf (take i xs) xs


ghci> quickCheck prop_prefix
+++ OK, passed 100 tests.
```

Example: Sort

What properties should hold for all inputs and outputs of a list sorting function?

Example: Sort

What properties should hold for all inputs and outputs of a list sorting function?

- The elements of the output should be in sorted order
- The output should have the same elements as the input

Example: Sort

What properties should hold for all inputs and outputs of a list sorting function?

- The elements of the output should be in sorted order
- The output should have the same elements as the input

Try to write these!

(The file already contains the functions sameElements and ordered.)

QuickCheck

```
prop_ordered :: [Int] -> Bool
prop_ordered xs = ordered (sort xs)

prop_elements :: [Int] -> Bool
prop_elements xs = sameElements (sort xs) xs


ghci> quickCheck prop_ordered
+++ OK, passed 100 tests.

ghci> quickCheck prop_elements
+++ OK, passed 100 tests.
```

QuickCheck

```
prop_ordered :: [Int] -> Bool
prop_ordered xs = ordered (sortBad xs)

prop_elements :: [Int] -> Bool
prop_elements xs = sameElements (sortBad xs) xs
```

QuickCheck

```
prop_ordered :: [Int] -> Bool
prop_ordered xs = ordered (sortBad xs)

prop_elements :: [Int] -> Bool
prop_elements xs = sameElements (sortBad xs) xs

ghci> quickCheck prop_ordered
+++ OK, passed 100 tests.

ghci> quickCheck prop_elements
*** Failed! Falsifiable (after 4 tests and 3 shrinks):
[0,0]
```

How does QuickCheck generate inputs?

The standard library contains a number of prebuilt *generators*.

- E.g. integers, lists of types for which it has generators
- For custom types, we must build our own generators
  - We will learn how to do this in a later lecture

Common property types

- Validity properties
- Postconditions
  - Roundtrip properties
  - Metamorphic properties
- Model-based properties

Validity properties

```
type SortedList = [Int]

insert :: Int -> SortedList -> SortedList
insert x xs = …

prop_insertOK :: Int -> SortedList -> Bool
prop_insertOK x xs = ordered (insert x xs)
```

Validity properties

```
type SortedList = [Int]

insert :: Int -> SortedList -> SortedList
insert x xs = …

prop_insertOK :: Int -> SortedList -> Bool
prop_insertOK x xs = ordered (insert x xs)
```

Do you see a problem with this property?

Validity properties

```
prop_insertOK :: Int -> SortedList -> Bool
prop_insertOK x xs = ordered (insert x xs)
```

```
ghci> quickCheck prop_insertOK
*** Failed! Falsifiable (after 4 tests and 3 shrinks):
2
[4,1]
```

Preconditions

```
prop_insertOK :: Int -> SortedList -> Bool
prop_insertOK x xs = ordered xs ==> ordered (insert x xs)
```

Postconditions

We often use PBT to test properties familiar from math.

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

Postconditions

We often use PBT to test properties familiar from math.

```
prop_negateInvolutive :: Int -> Bool
prop_negateInvolutive x = negate (negate x) == x
```

Postconditions

We often use PBT to test properties familiar from math.

```
prop_squareSqrtInverses :: Int -> Bool
prop_squareSqrtInverses x = sqrt (square x) == x
```

Postconditions

We often use PBT to test properties familiar from math.

```
prop_squareSqrtInverses :: Int -> Bool
prop_squareSqrtInverses x = sqrt (square x) == x
```

Properties that test the interaction of two functions are often called *metamorphic properties*.

Properties that test that composing two functions yields the initial input are called *round-trip properties*.

Model-Based Testing

We may have an existing implementation we want to compare ours to.

- E.g., an easy-to-understand version vs. an optimized version

```
prop_sortOpt :: [Int] -> Bool
prop_sortOpt xs = sortOpt xs == Data.List.sort xs
```

This style can be quicker to write, but only do it if you are truly confident the existing implementation is correct!

Comparison

Unit testing

- Must write examples by hand
- Examples generally do not fully specify the problem
- Examples are quick to write and generally intuitive

PBT

- Can test many examples at once
- Provides a clear mathematical specification of the program
- Can be less intuitive
- For many types, requires manually written generators

Comparison

Unit testing

- Quick iteration
- Testing specific cases
- Regression testing
- Functions that are their own specification

PBT

- Specifying core libraries
- Checking data structure invariants
- “Math-like” concepts
- Checking against a reference implementation

For further information

- How To Specify It! by John Hughes
- QuickCheck documentation
- Future lecture with more PBT details

Practice!