CS 360: Programming Languages
Lecture 13: (Property-Based) Testing

Geoffrey Mainland

Drexel University

Section 1

Administrivia

Administrivia

- Homework 7 will be released tomorrow.
- There will be no Gradescope tests—you must write your own tests.
- We will discuss the assignment in more detail at the end of lecture.

Section 2

Testing

Testing[1]

Image credit: Hamilton Richards via Wikipedia

"Program testing can be used to show the presence of bugs, but never to show their absence!"
—E. W. Dijkstra

[1]Thanks to John Hughes and Nick Smallbone at Chalmers for much of the following material.

Testing vs. Types

- Testing tells us something about *some* runs of a program.
- Types tell us something about *all* runs of a programs.
- Why not just rely on types?
- A type can only say so much. . .

```
reverse  :: [a] -> [a]
map      :: (a -> b) -> [a] -> [b]
square   :: Integer -> Integer
```

Testing vs. Proofs

- Proofs also tell us something about all runs of a programs, but proofs don't scale well (yet).
- Software verification is a very difficult problem and the subject of much research.

A Testing Quiz: What is...

- Test-driven development (TDD).
- Regression testing.
- Continuous integration (CI).
- Black-box testing.
- White-box testing.
- Property-based testing.

Hspec: Testing for Haskell

- The Hspec library provides easy mechanisms to define and run automated test suites.
- Inspired by the Ruby RSpec library.
- We've used it to provide you with tests for all Haskell-based assignments. For Racket assignments, we used RackUnit and rackcheck.
- Many unit testing libraries in many languages—when you have to write code in a new language, the first thing you should do is find a popular unit testing library for that language!

Hspec: An Example

```haskell
import Test.Hspec
import Test.QuickCheck
import Control.Exception (evaluate)

spec :: Spec
spec = do
  describe "Prelude.head" $ do
    it "returns the first element of a list" $
      head [23 ..] `shouldBe` (23 :: Int)

    it "returns the first element of an arbitrary list" $
      property $ \x xs -> head (x:xs) == (x :: Int)

    it "throws an exception if used with an empty list" $
      evaluate (head []) `shouldThrow` anyException
```

Hspec: An Example Run

```
> :load Spec.hs
[1 of 1] Compiling Main             ( Spec.hs,
    interpreted )
Ok, one module loaded.
*Main> hspec spec

Prelude.head
  returns the first element of a list
  returns the first element of an *arbitrary* list
    +++ OK, passed 100 tests.
  throws an exception if used with an empty list

Finished in 0.0010 seconds
3 examples, 0 failures
*Main>
```

Hspec: An Example

describe provides a name for a group of tests

shouldBe specifies what the result of a test should be.

[23..] is the list of all **Integers** from 23.

it labels a specific test.

```
spec :: Spec
spec = do
  describe "Prelude.head" $ do
    it "returns the first element of a list" $
      head [23..] `shouldBe` (23 :: Int)
    ⋮
```

Hspec: An Example

Defines a *property*-based test—we will see how to do this later.

```
spec :: Spec
spec = do
  describe "Prelude.head" $ do
    ⋮
    it "returns the first element of an arbitrary list" $
      property $ \x xs -> head (x:xs) == (x :: Int)
    ⋮
```

Hspec: An Example

```
spec :: Spec
spec = do
  describe "Prelude.head" $ do
    ⋮
    it "throws an exception if used with an empty list" $
      evaluate (head []) `shouldThrow` anyException
    ⋮
```

Forces evaluation.

shouldThrow checks that an exception is thrown.

Why do we need evaluate? Why can't we just check that head [] throws an exception?

What Happens When Tests Succeed?

```haskell
lookupTest :: Spec
lookupTest = it "lookup" $ lookup 1 [(1,"a")] `shouldBe` Just "a"
```

```
*Insert> hspec lookupTest

lookup

Finished in 0.0002 seconds
1 example, 0 failures
*Insert>
```

What Happens When Tests Fail?

```haskell
lookupTest :: Spec
lookupTest = it "lookup" $ lookup 1 [(1,"a")] `shouldBe` Just "b"
```

```
*Insert> hspec lookupTest

lookup FAILED [1]

Failures:

  Insert.hs:16:28:
  1) lookup
       expected: Just "b"
        but got: Just "a"

  To rerun use: --match "/lookup/"

Randomized with seed 718718230

Finished in 0.0006 seconds
1 example, 1 failure
*** Exception: ExitFailure 1
*Insert>
```

Section 3

Property-based Testing

Property-based Testing

- Idea: state logical properties in a "domain-specific language," automatically generate test cases from these properties.
- Originated in a Haskell library called QuickCheck (most cited paper from ICFP).
- Widely duplicated in other languages. The hypothesis library for Python is one such implementation.

A Straightforward Fibonacci

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

A Fast Fibonacci

```
badfib :: Integer -> Integer
badfib n = aux n 0 1
  where
    aux 0 _ b = b
    aux n a b = aux (n - 1) b (a + b)
```

```
*Main> badfib 0
1
*Main> badfib 1
1
*Main>
```

A Fast Fibonacci: Take 2

```
fib :: Integer -> Integer
fib n = aux n 0 1
  where
    aux 0 a _ = a
    aux n a b = aux (n - 1) b (a + b)
```

```
*Main> fib 0
0
*Main> fib 1
1
*Main>
```

Testing Fibonacci

```
*Main> fib 0
0
*Main> fib 1
1
*Main> fib 2
1
*Main> fib 3
2
*Main> fib 4
3
*Main>
```

- Works for numbers 0–4!
- Are you convinced?
- Can you write a function that passes these “unit tests” but is not a correct implementation of Fibonacci?

Our Property

```
prop_fibn :: Integer -> Bool
prop_fibn n = fib n == fib (n-1) + fib (n-2)
```

- Question: Are we done?

Some More Testing

```
*Main> prop_fibn 2
True
*Main> prop_fibn 3
True
*Main> prop_fibn 4
True
*Main>
```

- Looks good! How about automating it!

Property-based Testing

```
import Test.QuickCheck

*Main> quickCheck prop_fibn
```

Fix the Property

```
prop_fibn n = n > 1 ==> fib n == fib (n-1) + fib (n-2)
```

- Restrict tests to numbers > 1.
- The (==>) operator discards the test immediately if the predicate is false.

Run Quickcheck

```
*Main> quickCheck prop_fibn
+++ OK, passed 100 tests.
*Main>
```

- The property passed 100 random tests.
- That is, the following was true for 100 random tests where $n > 1$: `fib n == fib (n-1) + fib (n-2)`

What values were tested?

$2, 2, 2, 2, 2, 4, 2, 2, 4, 5, 8, 4, 14, 6, 11, 7, 3, 12, 15, 20, 15, 21, 14, \ldots$

- Start with generating small numbers.
- Gradually increase their size.

Is it enough?

```
prop_fibn n = n > 1 ==> fib n == fib (n-1) + fib (n-2)
```

This property only checks whether the results are correct relative to each other. The following sequences all satisfy it:

- `[0,1,1,2,3,5,8,13,21,34,...`
- `[4,7,11,18,29,47,76,123,199,322,...`
- `[0,0,0,0,0,0,0,0,0,0,...`

How to fix that?

```
prop_fibn n = n > 1 ==> fib n == fib (n-1) + fib (n-2)
```

Add new properties:

```
prop_fib0 = fib 0 == 0
prop_fib1 = fib 1 == 1
```

How sure can we be?

- Testing these properties is in some sense complete.
- We test base cases.
- We test some number of steps.
- Each step has a chance of being tested.
- Every bug has a chance of showing up, but we might not reach it if it is far away.
- If the logical properties hold, the implementation *must* be correct. The catch: we're not guaranteed that the logical properties are tested with all possible inputs.

How sure can we be? Quite sure!

Testing Quicksort

```
qsort :: Ord a => [a] -> [a]
qsort []     =  []
qsort (x:xs) =  qsort (filter (< x) xs) ++
                [x] ++
                qsort (filter (> x) xs)
```

Testing Quicksort

```
*Main> qsort []
[]
*Main> qsort [1]
[1]
*Main> qsort [2,4,1]
[1,2,4]
*Main> qsort [6,2]
[2,6]
*Main> qsort [10,9,8,4]
[4,8,9,10]
*Main> qsort [3,5,6]
[3,5,6]
*Main>
```

Looks good!

Some properties...

```haskell
isSorted :: Ord a => [a] -> Bool
isSorted []  = True
isSorted [_] = True
isSorted (x:ys@(y:_))
  | x <= y    = isSorted ys
  | otherwise = False

prop_qsort1 :: [Int] -> Bool
prop_qsort1 l = isSorted $ qsort l

prop_qsort2 :: [Int] -> Bool
prop_qsort2 l =
  qsort l == qsort (reverse l)

prop_qsort3 :: [Int] -> [Int] -> Bool
prop_qsort3 l1 l2 =
  qsort (l1 ++ l2) == qsort (l2 ++ l1)
```

Note the type signatures. Why are they needed?

Testing the properties...

```
*Main> quickCheck prop_qsort1
+++ OK, passed 100 tests.
*Main> quickCheck prop_qsort2
+++ OK, passed 100 tests.
*Main> quickCheck prop_qsort3
+++ OK, passed 100 tests.
*Main>
```

One more property...

```
prop_qsort4 :: [Int] -> Bool
prop_qsort4 l =
  length (qsort l) == length l
```

```
*Main> quickCheck prop_qsort4
*** Failed! Falsifiable (after 4 tests and 1 shrink):
[3,3]
```

- Oops...

What happened?

```
*Main> length [3,3]
2
*Main> length (qsort [3,3])
1
*Main> qsort [3,3]
[3]
*Main>
```

Back to the code

```haskell
qsort :: Ord a => [a] -> [a]
qsort []     =  []
qsort (x:xs) =  qsort (filter (< x) xs) ++
                [x] ++
                qsort (filter (> x) xs)
```

Lessons for us

- We should test sorting functions with repeated elements (now we know...).
- Auto-generated data gives us test cases we wouldn't necessarily think of otherwise.

Generators

```haskell
class Arbitrary a where
    arbitrary :: Gen a
    shrink :: a -> [a]
```

- `arbitrary` generates a "random" value of type a.
- `shrink` attempts to find smaller values "like" its argument—useful for finding a minimal counter-example.
- The `arbitrary` function is like the `build` function from the Generative Art homework, but embodied in a type class.
- The **`Gen`** type is a **monad**. It provides a random source (like **`RandomDouble`**) and access to a size parameter (like depth) that can be used to control the size of generated values.

Sampling from Generators

```
arbitrary :: Arbitrary a => Gen a
```

We can see what the default string generator chooses in ghci:

```
*Main> sample (arbitrary :: Gen String)
""
""
"\221U"
"z\SUB\ESC"
"PlQwq"
"\220bu"
"0\139)\182\&7]U%"
"\NAK\156l\SUB"
""
":\207)jI\SI\245"
"\t7O%\STXe[\SYN5\\L\155\&18I\SOH{q\231"
*Main>
```

Generators

- Provide a way to automate test-case generation.
- Quality of test cases depends on the quality of the generator!
- When a test case fails, `shrink` will be used to search smaller test cases for failure.
- Keep searching smaller test cases until no more failure.
- Hopefully leads to a *minimal* failing test case.
- Down-side: need to write a “smart” generator.

Summary: Property-based Testing

- Unit tests are straightforward to write and understand.
- Property-based testing *generates* tests and can cover many more test cases automatically—including some the test writer may not have thought of.
- A property is an *invariant*. Property-based testing tests that the invariant holds for many examples, but it does not *prove* that the invariant holds for all cases.
- Writing a good generator is critical—and usually what requires the most work.

Section 4

Interval Sets

Interval Set Representation

- Simple representation of sets: sorted list without duplicates.
- Much more efficient representation for sets of integers: a sorted list of (inclusive) ranges.
- Consider the set $\{2, 3, 4, 10, 11, 12, 13\}$:
  - Haskell representation as a sorted list: `[2,3,4,10,11,12,13]`
  - Haskell representation as an interval set: `[(2,4), (10,13)]`
- The type of interval sets is simple:

```
newtype IntSet = IntSet [(Int, Int)]
  deriving (Eq, Ord, Show)
```

Interval Sets: Asymptotic Complexity

Operation	Asymptotic complexity
`empty`	$O(1)$
`member`	$O(n)$ in the number of intervals
`delete`	$O(n)$ in the number of intervals
`insert`	$O(n)$ in the number of intervals
`merge`	$O(n + m)$ in the number of intervals in the two sets

- Good: Implement `insert` in terms of `merge`.
- Bad: Implement `merge` in terms of `insert`.
- What is the (likely) asymptotic complexity of an implementation of `merge` in terms of `insert`?

Interval Set Invariants

1. For any interval (x, y) in a set, x should not be greater than y (so the intervals are not empty). **Example**: $(2, 5)$ and $(4, 4)$ are valid intervals, but $(7, 2)$ is not.
2. Two intervals in the same set should not be overlapping or touching—two such intervals must instead be represented by a single interval. **Examples**: The intervals $(2, 6)$ and $(4, 10)$ should be represented by a single interval $(2, 10)$. The intervals $(3, 6)$ and $(7, 11)$ should be represented by a single interval $(3, 11)$.
3. The intervals in a given interval set should occur in ascending order. **Example**: `[(1,3),(5,9),(15,16)]` is a valid interval set, but `[(5,10),(1,3)]` is not.

These invariants guarantee a **canonical** representation for any set of integers.

Capturing Interval Set Invariants as Properties

1. For any interval (x, y) in a set, x should not be greater than y (so the intervals are not empty).
2. Two intervals in the same set should not be overlapping or touching—two such intervals must instead be represented by a single interval.
3. The intervals in a given interval set should occur in ascending order.

```
invariant :: IntSet -> Bool
invariant (IntSet xs) =
    all (\(lo,hi) -> lo <= hi) xs &&
    all (\((_,hi), (lo,_)) -> lo > hi + 1) (xs `zip` drop 1 xs)
```

Checking Properties of Interval Sets

- We will **model** interval sets as a sorted list of numbers with no duplicates. This is not efficient, but it is easy to work with!
- We state properties of interval sets by relating operations on interval sets to equivalent operations *on the model*.

```
model :: IntSet -> [Int]
model (IntSet xs) = nub $ sort $ concat [[lo..hi] | (lo,hi) <- xs]
```

Model-based Testing

- Model-based testing tests an implementation against a **model**.
- The model should be simple. It's OK if it isn't efficient!
- The model could also be another, independent implementation.
- Model-based approach:
  1. Perform operation on original representation, find model of result.
  2. Perform equivalent operation *directly on the model of the original representation*. This gives us a second model.
  3. Test that the two models are equivalent.
- Extremely powerful technique for property-based testing.

Using the Model

- How should we state a property about the `member` function that relates an operation on an interval sets to an operation on its model?
- Hint: `member x xs == ...`

Testing as Adversarial Game

```
prop_delete :: Int -> IntSet -> Bool
prop_delete x xs = notElem x (model (delete x xs))
```

- What do you think of this property?
- Pretend you are not a bad programmer, but an *evil* programmer—you want to pass the test suite with an intentionally incorrect implementation, ideally doing as little work as possible.
- How would an evil programmer implement `delete`?
- How can you protect against the evil programmer by using a property-based test relating interval sets to their models?
- Need to check not only that `x` is no longer in the set, but that we haven't added any additional elements and that we haven't removed any other elements.
- We can do both by relating the *model of our result* to the model we get by deleting an element from the *model of our input*.

Model Example

Delete 5 from `[(1,6)]`.

- Should give us `[(1,4), (6,6)]`
- Model of `[(1,4), (6,6)]` is `[1,2,3,4,6]`
- Model of input is `[1,2,3,4,5,6]`.
- Deleting 5 from `[1,2,3,4,5,6]` gives `[1,2,3,4,6]`, so the models match!
- But we want to state a general property, not provide a single unit test. You should generalize this example to a property that holds for *all* values of `x` and `xs` when we evaluate `delete x xs`.

A flaw with the model-based approach

- Because the model is a list of integers, i.e., it “blows up” the intervals, it cannot efficiently handle large sets.
- Use unit tests to make sure your implementation correctly handles large interval sets!

Section 5

Simplifying Expressions

Simplify the Following Expressions

Assume e is well-typed and evaluation of e terminates, i.e., e does not "loop forever" or throw an error.

Simplify the following expressions:

- `if e then True else False`
- `e == True`
- ```
  f x | g x == True = True
      | otherwise   = False
  ```
- `\x -> f x`
- `[`e_1`] ++ `e_2
- ```
  case e of
    Nothing -> Nothing
    Just x  -> Just x
  ```