Recall Unit Tests

- We started off the semester by talking about testing.
  
  What is a test?
Recall Unit Tests

• We started off the semester by talking about testing.
  – Input to drive a behavior
  – An oracle to check a behavior
Recall Unit Tests

• We started off the semester by talking about testing.
  – Input to drive a behavior
  – An oracle to check a behavior
Recall Unit Tests

- We started off the semester by talking about testing.
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing *samples* the concrete behaviors of a program
Recall Unit Tests

• We started off the semester by talking about testing.
  – Input to drive a behavior
  – An oracle to check a behavior

• Testing samples the concrete behaviors of a program

Did we have ways of getting more information from each test?
Recall Unit Tests

- We started off the semester by talking about testing.
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing *samples* the concrete behaviors of a program
  - Analyzing equivalence classes
Recall Unit Tests

• We started off the semester by talking about testing.
  – Input to drive a behavior
  – An oracle to check a behavior

• Testing samples the concrete behaviors of a program
  – Analyzing equivalence classes
Recall Unit Tests

• We started off the semester by talking about testing.
  – Input to drive a behavior
  – An oracle to check a behavior

• Testing samples the concrete behaviors of a program
  – Analyzing equivalence classes
  – Program analysis can find richer bugs over a test suite.
Recall Unit Tests

• We started off the semester by talking about testing.
  – Input to drive a behavior
  – An oracle to check a behavior

• Testing samples the concrete behaviors of a program
  – Analyzing equivalence classes
  – Program analysis can find richer bugs over a test suite.

Do these completely solve the problem?
Recall Unit Tests

• We started off the semester by talking about testing.
  – Input to drive a behavior
  – An oracle to check a behavior

• Testing *samples* the concrete behaviors of a program
  – Analyzing equivalence classes
  – Program analysis can find richer bugs over a test suite.

• Formal reasoning & program analysis can also make each test cover more behavior!
Recall Unit Tests

- We started off the semester by talking about testing.
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing samples the concrete behaviors of a program
  - Analyzing equivalence classes
  - Program analysis can find richer bugs over a test suite.

- Formal reasoning & program analysis can also make each test cover more behavior!
  - Property based testing
Abstracting Unit Tests

TEST(testCaseName, testName) {
    // Set up scenario
    // Run scenario on component
    // Check oracle
}
Abstracting Unit Tests

TEST(testCaseName, testName) {
  // Set up scenario
  // Run scenario on component
  // Check oracle
}

• A scenario could be concrete or abstract

  \[ x = 5 \quad \forall x : x > 0 \]
Abstracting Unit Tests

- A scenario could be **concrete** or **abstract**
  - $x = 5$  
  - $\forall x : x > 0$

- For an abstract test case, we could (1) generate tests and (2) check the oracle
  - Emphasis is on the scenario & oracle
Abstracting Unit Tests

- A scenario could be **concrete** or **abstract**

\[ x = 5 \quad \forall x : x > 0 \]

- For an abstract test case, we could (1) generate tests and (2) check the oracle
  - Emphasis is on the scenario & oracle

How can we generate tests?
Abstracting Unit Tests

- A scenario could be **concrete** or **abstract**

\[ x = 5 \quad \forall x : x > 0 \]

- For an abstract test case, we could (1) generate tests and (2) check the oracle
  - Emphasis is on the scenario & oracle
- 2 approaches we have already seen can be used
  1) Random testing
  2) Symbolic execution
Property Based Testing

- This forms the motivation of *property based testing*
Property Based Testing

- This forms the motivation of *property based testing*
  - Testing that focuses on functional properties and generates many tests to check them.
Property Based Testing

- This forms the motivation of *property based testing*
  - Testing that focuses on functional properties and generates many tests to check them.
- Definition is still evolving
  - Originated with QuickCheck for Haskell in 2000
  - Focus was on generating many random tests from rich type information and checking property assertions
Property Based Testing

• This forms the motivation of *property based testing*
  – Testing that focuses on functional properties and generates many tests to check them.

• Definition is still evolving
  – Originated with QuickCheck for Haskell in 2000
  – Focus *was* on generating many random tests from rich type information and checking property assertions
  – Test case reduction was also automatically applied
Property Based Testing

- This forms the motivation of **property based testing**
  - Testing that focuses on functional properties and generates many tests to check them.

- Definition is still evolving
  - Originated with QuickCheck for Haskell in 2000
  - Focus *was* on generating many random tests from rich type information and checking property assertions
  - Test case reduction was also automatically applied
  - Now includes symbolic execution
Property Based Testing

- Traditional testing can be seen as *example based*.

\[
x = 5
\]
Property Based Testing

- Traditional testing can be seen as *example based*.
- Property based testing focuses on the generic properties that should hold.

\[ \forall x : x > 0 \]
Property Based Testing

- Traditional testing can be seen as *example based*.
- Property based testing focuses on the generic properties that should hold.

\[ \forall x : x > 0 \]

What is $x$ and how does it fit into testing?
Property Based Testing

- Traditional testing can be seen as *example based*.
- Property based testing focuses on the generic properties that should hold.
  \[ \forall x : x > 0 \]
- For random testing, *generators* can provide a way to randomly sample complex types.
Property Based Testing

- Traditional testing can be seen as *example based*.
- Property based testing focuses on the generic properties that should hold.
  \[ \forall x : x > 0 \]
- For random testing, *generators* can provide a way to randomly sample complex types.
  - Substantial effort to create generator infrastructure initially
Property Based Testing

- Follow common test patterns:
  - Symmetry
    \[ \text{encode(decode(x))} == x \]
Property Based Testing

• Follow common test patterns:
  – Symmetry    \( \text{encode}(\text{decode}(x)) == x \)
  – Alternatives \( \text{bubbleSort}(x) == \text{qsort}(x) \)
Property Based Testing

• Follow common test patterns:
  – Symmetry: \( \text{encode(decode(x)) == x} \)
  – Alternatives: \( \text{bubbleSort(x) == qsort(x)} \)
  – Induction: \( \text{car(cons(head, tail)) == head} \)
Property Based Testing

- Follow common test patterns:
  - Symmetry: $\text{encode(decode(x)) == x}$
  - Alternatives: $\text{bubbleSort(x) == qsort(x)}$
  - Induction: $\text{car(cons(head,tail)) == head}$
  - Idempotence: $\text{qsort(qsort(x)) == qsort(x)}$
Property Based Testing

- Follow common test patterns:
  - Symmetry: $\text{encode(\text{decode}(x))} == x$
  - Alternatives: $\text{bubbleSort}(x) == \text{qsort}(x)$
  - Induction: $\text{car(cons(head, tail))} == \text{head}$
  - Idempotence: $\text{qsort(\text{qsort}(x))} == \text{qsort}(x)$
  - Invariants: $\text{qsort}(x).\text{size()} == x.\text{size()}$
Property Based Testing

- Follow common test patterns:
  - Symmetry \( \text{encode} \left( \text{decode} (x) \right) = x \)
  - Alternatives \( \text{bubbleSort} (x) = \text{qsort} (x) \)
  - Induction \( \text{car} \left( \text{cons} (\text{head}, \text{tail}) \right) = \text{head} \)
  - Idempotence \( \text{qsort} \left( \text{qsort} (x) \right) = \text{qsort} (x) \)
  - Invariants \( \text{qsort} (x). \text{size} () = x. \text{size} () \)

What else might we check here?
Benefits of PBT

• Tests can have a clear, mathematical presentation
Benefits of PBT

• Tests can have a clear, mathematical presentation

• Can avoid finding & writing every case for each property (focus on the what not the how)
Benefits of PBT

- Tests can have a clear, mathematical presentation
- Can avoid finding & writing *every case* for each property *(focus on the what not the how)*
- Can decrease maintenance costs with the same (& sometime greater) coverage
Benefits of PBT

- Tests can have a clear, mathematical presentation
- Can avoid finding & writing every case for each property (focus on the what not the how)
- Can decrease maintenance costs with the same (& sometime greater) coverage

Random testing often gives these in practice. Is that a guarantee?
In Practice: Hypothesis

- Hypothesis (https://hypothesis.works/)
  - Python, Java, (speculative C, C++)
  - Random testing approach (maybe SymEx in future)
  - Uses Generators to construct data
In Practice: Hypothesis

- **Hypothesis** ([https://hypothesis.works/](https://hypothesis.works/))
  - Python, Java, (speculative C, C++)
  - Random testing approach (maybe SymEx in future)
  - Uses Generators to construct data

```python
from hypothesis import given
from hypothesis.strategies import text

given(text())
@example('')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```
In Practice: Hypothesis

- Hypothesis (https://hypothesis.works/)
  - Python, Java, (speculative C, C++)
  - Random testing approach (maybe SymEx in future)
  - Uses Generators to construct data

```python
from hypothesis import given
from hypothesis.strategies import text

given(text())
@example('')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```
In Practice: Hypothesis

- **Hypothesis** ([https://hypothesis.works/](https://hypothesis.works/))
  - Python, Java, (speculative C, C++)
  - Random testing approach (maybe SymEx in future)
  - Uses Generators to construct data

```python
from hypothesis import given
from hypothesis.strategies import text

given(text())
@example('')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```
In Practice: Hypothesis

- Hypothesis (https://hypothesis.works/)
  - Python, Java, (speculative C, C++)
  - Random testing approach (maybe SymEx in future)
  - Uses Generators to construct data

```python
from hypothesis import given
from hypothesis.strategies import text
given(text())
@example('')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```
In Practice: Hypothesis

- Many generators are built in.
In Practice: Hypothesis

- Many generators are built in.
- Complex input spaces may require custom generators
In Practice: Hypothesis

- Many generators are built in.
- Complex input spaces may require custom generators

```python
@composite
def distinct_strings_with_common_characters(draw):
    x = draw(text(), min_size=1)
    y = draw(text(alphabet=x))
    assume(x != y)
    return (x, y)
```

- A rich set of primitives is available for more complex generator needs
In Practice: DeepState

- DeepState (https://github.com/trailofbits/deepstate)
  - C and C++ focused
  - API is compatible with GoogleTest
  - Symbolic execution tries to automatically extract inputs
In Practice: DeepState

- DeepState (https://github.com/trailofbits/deepstate)
  - C and C++ focused
  - API is compatible with GoogleTest
  - Symbolic execution tries to automatically extract inputs

```c
TEST(PrimePolynomial, OnlyGeneratesPrimes_NoStreaming) {
    symbolic_unsigned x, y, z;
    DeepState_Assume(x > 0);
    unsigned poly = (x * x) + x + 41;
    DeepState_Assume(y > 1);
    DeepState_Assume(z > 1);
    DeepState_Assume(y < poly);
    DeepState_Assume(z < poly);
    DeepState_Assert(poly != (y * z));
    DeepState_Assert(IsPrime(Pump(poly)));
}
```
Summary: Property Based Testing

- An approach for testing based on the intended properties rather than the implementation
Summary: Property Based Testing

- An approach for testing based on the intended properties rather than the implementation
- Still tries to cover the behaviors of the implementation as well
Summary: Property Based Testing

- An approach for testing based on the intended properties rather than the implementation
- Still tries to cover the behaviors of the implementation as well
- Availability improves every year