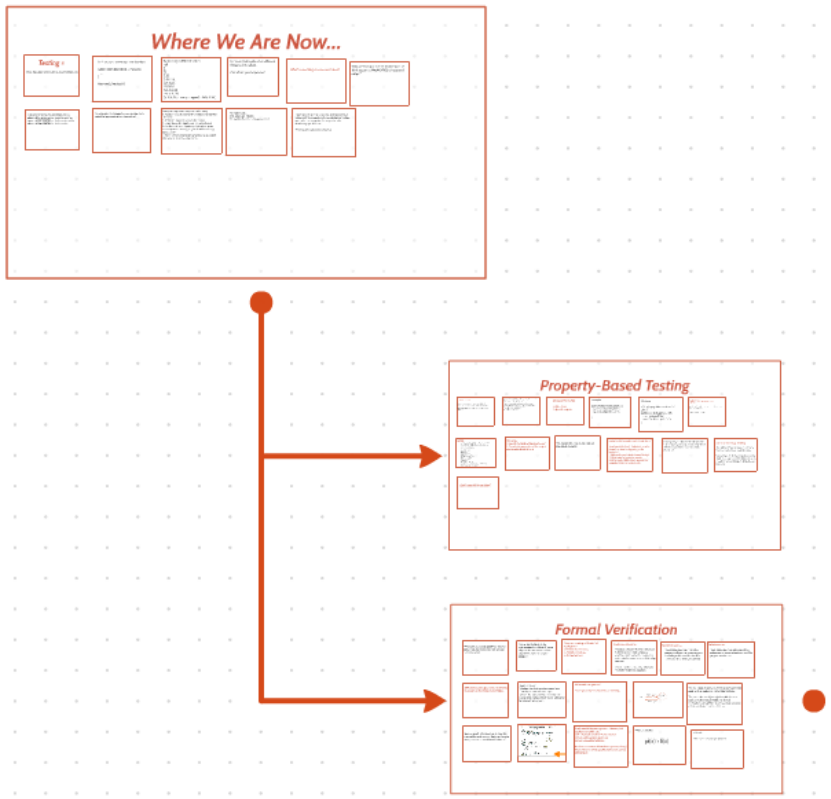
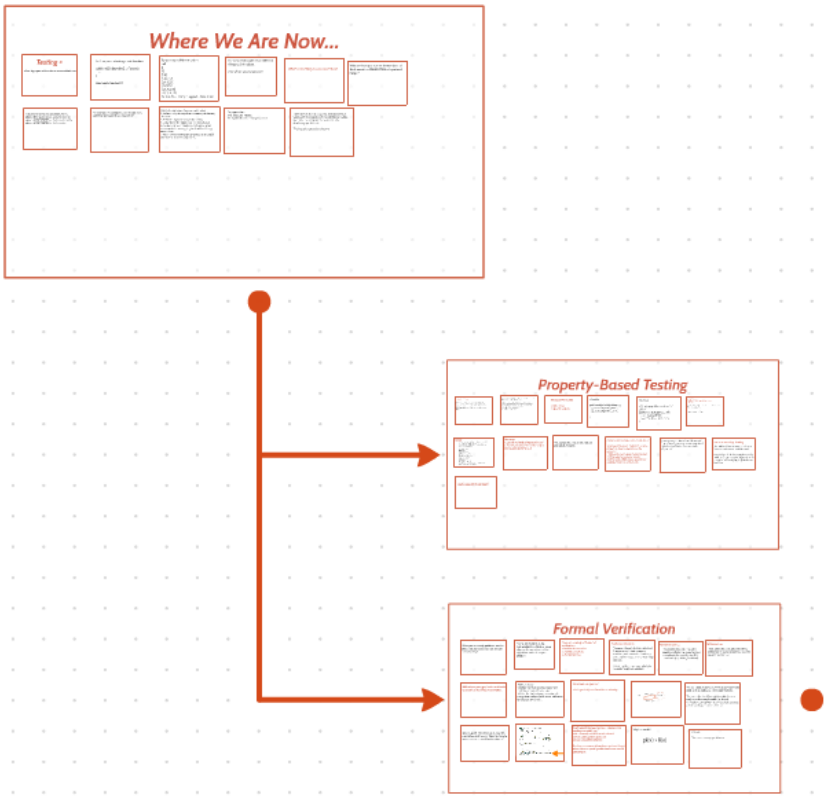
Property-Based Testing and Formal Verification



Property-Based Testing and Formal Verification



Where We Are Now...

Testing =

Checking expected behavior vs observed behavior

Let's say we are testing a sort function.

```
public int[] billSort(int[] arrToSort) {
    ....
}
```

How would we test it?

Try passing in different values:
null
[]
[1]
[1,2]
[1,2,3,4,5]
[5,4,3,2,1]
[0,0,0,0,0]
[9,3,4,2,1,6]
[-9, 2, 4, -3]
[9, 4, 2, 19, ... (many integers) ... 982, 4, 23]

But wow, that's quite a few different things to think about.

And what if you forget one?

What's something else we could check?

Why not hop up a level of abstraction and think about the PROPERTIES of input and output?

Instead of checking expected behavior vs observed behavior per se, we can check the expected PROPERTIES of the behavior vs the observed PROPERTIES of the behavior.

For example, what properties do we expect of a sorted list compared to a list passed in?

1. Output array same size as passed-in array
2. Values in output array always increasing or staying the same
3. Value in output array never decreasing
4. Every element in input array is in output array
5. No element not in input array is in output array
6. Idempotent - running it again should not change output array
7. Pure - running it twice on same input array should always result in same output array

For input values:
1. All values are integers
2. Length is 0 or more integers, or null

Now that we have the properties of expected input values, and the properties of expected output values, we can let the computer do the grunt work of developing specific tests.

This is called *property-based testing*.

Testing =

Checking expected behavior vs observed behavior

Let's say we are testing a sort function.

```
public int[] billSort(int[] arrToSort) {
    ....
}
```

How would we test it?

Try passing in different values:

null

[]

[1]

[1,2]

[1,2,3,4,5]

[5,4,3,2,1]

[0,0,0,0]

[9,3,4,2,1,6]

[-9, 2, 4, -3]

[9, 4, 2, 19, ... (many integers) ... 982, 4, 23]

But wow, that's quite a few different things to think about.

And what if you forget one?

What's something else we could check?

Why not hop up a level of abstraction and think about the PROPERTIES of input and output?

Instead of checking expected behavior vs observed behavior per se, we can check the expected PROPERTIES of the behavior vs the observed PROPERTIES of the behavior.

For example, what properties do we expect of a sorted list compared to a list passed in?

1. Output array same size as passed-in array
2. Values in output array always increasing or staying the same
3. Value in output array never decreasing
4. Every element in input array is in output array
5. No element not in input array is in output array
6. Idempotent - running it again should not change output array
7. Pure - running it twice on same input array should always result in same output array

For input values:

1. All values are integers
2. Length is 0 or more integers, or null

Now that we have the properties of expected input values, and the properties of expected output values, we can let the computer do the grunt work of developing specific tests.

This is called *property-based testing*.

Property-Based Testing

A New Kind of Testing

Presented at ICFP 2000 in the paper "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs"

http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf

Much more popular in functional programming world than the imperative world, for a variety of reasons

1. Pure functional code is much easier to test in this manner
2. Organizations using imperative languages less familiar with it
3. Requires some mathematical background
4. Considered "overkill" by some
5. Considered "weird" by some

Not useful for testing:

1. Side effects
2. Specific outputs

Example:

```
public void printGlobalStats() {
  System.out.print("Stat 1");
  System.out.println(_stat1);
  ...
}
```

Example:

```
// Login page title should be "hi"
@Test
public void testLoginSaysHi() {
  title = page.getTitle();
  assertTrue(title.equals("hi"));
}
```

Useful for testing:
A variety of inputs --> specific kinds of outputs
Pure code

Since Haskell is "purely functional", with no side effects of any kind*, this is a perfect match.

* Outside of monads, of course.

NOTE:

The concept originated in functional programming but is still useful in imperative languages! There are a variety of QuickCheck-like testing solutions out there in different languages, e.g.

Java: junit-quickcheck
Ruby: rantly
Scala: scalacheck
Python: pytest-quickcheck
Node.js: node-quickcheck
Clojure: simple-check
C++: QuickCheck++
.NET: FsCheck
Erlang: Erlang/QuickCheck
The only one I couldn't find is a version for PHP. A good term project for someone, perhaps?

Two steps:
1. Specify the kinds of inputs allowed
2. Specify the properties of the output that should ALWAYS hold

These properties that always hold are also called *invariants*.

QuickCheck then makes our test suite for us!

Think about the levels of abstraction we've jumped up since the beginning of the semester:
1. Write and execute tests (manual testing)
2. Write tests, let computer execute
3. Write what KINDS of tests we want, let computer write tests and execute

This is gaining traction outside the word of functional programming, as some aspects of FP leak into production systems (Scala, Clojure, etc.)

One more cool thing.. shrinking

If QuickCheck finds an issue, it will try to find the smallest version of that issue.

Say sorting with 4s doesn't work correctly. It will try to figure out that [4] breaks, even though it first found [5,6,2,1,4] falsifies an invariant.

Let's see it in action!

A New Kind of Testing

Presented at ICFP 2000 in the paper "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs"

http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf

Much more popular in functional programming world than the imperative world, for a variety of reasons

1. Pure functional code is much easier to test in this manner
2. Organizations using imperative languages less familiar with it
3. Requires some mathematical background
4. Considered "overkill" by some
5. Considered "weird" by some

Not useful for testing:

1. Side effects
2. Specific outputs

Example:

```
public void printGlobalStats() {
    System.out.print(''Stat 1'');
    System.out.println(_stat1);
    ...
}
```

Example:

```
// Login page title should be "hi"
@Test
public void testLoginSaysHi() {
   title = page.getTitle();
   assertTrue(title.equals("hi"));
}
```

Useful for testing:
A variety of inputs -> specific kinds of outputs
Pure code

Since Haskell is "purely functional", with no side effects of any kind*, this is a perfect match.

* Outside of monads, of course.

NOTE:

The concept originated in functional programming but is still useful in imperative languages! There are a variety of QuickCheck-like testing solutions out there in different languages, e.g.

Java: junit-quickcheck
Ruby: rantly
Scala: scalacheck
Python: pytest-quickcheck
Node.js: node-quickcheck
Clojure: simple-check
C++: QuickCheck++
.NET: FsCheck
Erlang: Erlang/QuickCheck
The only one I couldn't find is a version for PHP. A good term project for someone, perhaps?

Two steps:

1. Specify the kinds of inputs allowed

2. Specify the properties of the output that should ALWAYS hold

These properties that always hold are also called *invariants*.

QuickCheck then makes our test suite for us!

Think about the levels of abstraction we've jumped up since the beginning of the semester:

1. Write and execute tests (manual testing)

2. Write tests, let computer execute

3. Write what KINDS of tests we want, let computer write tests and execute

This is gaining traction outside the word of functional programming, as some aspects of FP leak into production systems (Scala, Clojure, etc.)

One more cool thing.. shrinking

If QuickCheck finds an issue, it will try to find the smallest version of that issue.

Say sorting with 4s doesn't work correctly. It will try to figure out that [4] breaks, even though it first found [5,6,2,1,4] falsifies an invariant.

Let's see it in action!

Formal Verification

When you absolutely, positively need to prove that your code is correct, you can formally verify it.

Formal verification is using mathematical models to prove or disprove the correctness of the algorithms and code of your program.

There are a variety of "levels" of verification -
1. Predictable execution
2. Partial correctness
3. Full correctness

Predictable Execution

The code is free of what we would call "runtime errors"; that is, no array overflows, null pointer dereferencing, uninitialized variable access, division by zero, etc.

NB not that these are very unlikely, but PROVEN TO BE IMPOSSIBLE.

Partial Correctness

Predictable execution, PLUS the program will give the correct answer according to the specification IF it terminates (yay Halting Problem!).

Full Correctness

Predictable execution, plus everything referenced in partial correctness, plus the program terminates.

Difficulty to prove gets harder and harder as you go up the levels of verification.

How is it done?
1. Before-the-fact: construct code from formally-proven subset of logic
2. After-the-fact: attempt to verify code using static analysis (much easier with some languages than others...)

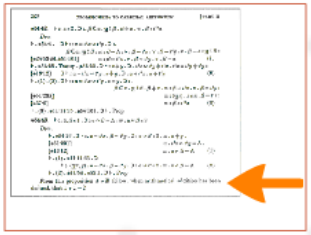
What tools can you use?

You're probably familiar with one already...

There are tools to convert a Finite State Machine to code, or show code as a Finite State Machine.

There are other tools/descriptions which can be used as mathematical models for formal verification, such as Petri nets, Hoare logic, process calculus, and operational semantics.

Sounds great! Why don't we just use this everywhere and the only things we have to worry about are the odd meteor impact?



Really overkill for most systems. However, it is used commercially, e.g.:
seL4 - Formally verified OS microkernel
Various cryptographic protocols
Various embedded software

Much more common in hardware systems, though, where I/O is very well specified and errors can be catastrophic

Why is it overkill?

$$pi(x) > li(x)$$

It's hard.

There are so many special cases.

When you absolutely, positively need to prove that your code is correct, you can formally verify it.

Formal verification is using mathematical models to prove or disprove the correctness of the algorithms and code of your program.

There are a variety of "levels" of verification -

1. Predictable execution

2. Partial correctness

3. Full correctness

Predictable Execution

The code is free of what we would call "runtime errors"; that is, no array overflows, null pointer dereferencing, uninitialized variable access, division by zero, etc.

NB not that these are very unlikely, but PROVEN TO BE IMPOSSIBLE.

Partial Correctness

Predictable execution, PLUS the program will give the correct answer according to the specification IF it terminates (yay Halting Problem!).

Full Correctness

Predictable execution, plus everything referenced in partial correctness, plus the program terminates.

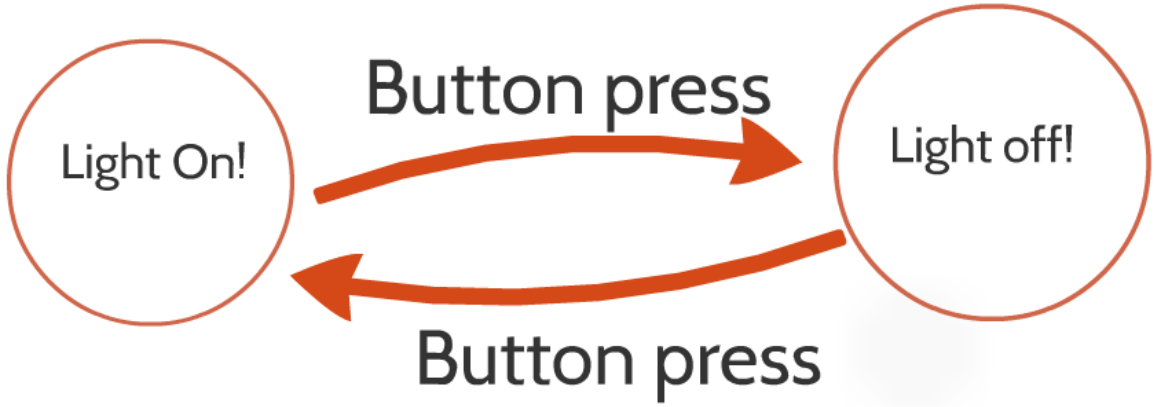
Difficulty to prove gets harder and harder as you go up the levels of verification.

How is it done?

1. Before-the-fact: construct code from formally-proven subset of logic
2. After-the-fact: attempt to verify code using static analysis (much easier with some languages than others...)

What tools can you use?

You're probably familiar with one already...



Light On!


Light off!

There are tools to convert a Finite State Machine to code, or show code as a Finite State Machine.

There are other tools/descriptions which can be used as mathematical models for formal verification, such as Petri nets, Hoare logic, process calculus, and operational semantics.

Sounds great! Why don't we just use this everywhere and the only things we have to worry about are the odd meteor impact?

***54·42.** $\vdash :: \alpha \in 2 . \supset :. \beta \subset \alpha . \exists ! \beta . \beta \neq \alpha . \equiv . \beta \in \iota''\alpha$

Dem.

$\vdash . *54·4 . \quad \supset \vdash :: \alpha = \iota'x \cup \iota'y . \supset :.$

$\beta \subset \alpha . \exists ! \beta . \equiv : \beta - \Lambda . \vee . \beta = \iota'x . \vee . \beta = \iota'y . \vee . \beta = \alpha : \exists ! \beta :$

[*24·53·56.*51·161] $\equiv : \beta = \iota'x . \vee . \beta = \iota'y . \vee . \beta = \alpha$ (1)

$\vdash . *54·25 . \text{Transp} . *52·22 . \supset \vdash : x \neq y . \supset . \iota'x \cup \iota'y \neq \iota'x . \iota'x \cup \iota'y \neq \iota'y :$

[*13·12] $\supset \vdash : \alpha = \iota'x \cup \iota'y . x \neq y . \supset . \alpha \neq \iota'x . \alpha \neq \iota'y$ (2)

$\vdash . (1) . (2) . \supset \vdash :: \alpha = \iota'x \cup \iota'y . x \neq y . \supset :.$

$\beta \subset \alpha . \exists ! \beta . \beta \neq \alpha . \equiv : \beta = \iota'x . \vee . \beta = \iota'y :$

[*31·235] $\equiv : (\exists x) . x \in \alpha . \beta = \iota'x :$

[*37·6] $\equiv : \beta \in \iota''\alpha$ (3)

$\vdash . (3) . *11·11·35 . *54·101 . \supset \vdash . \text{Prop}$

***54·43.** $\vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$\vdash . *54·26 . \supset \vdash :. \alpha = \iota'x . \beta = \iota'y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$

[*51·231] $\equiv . \iota'x \cap \iota'y = \Lambda .$

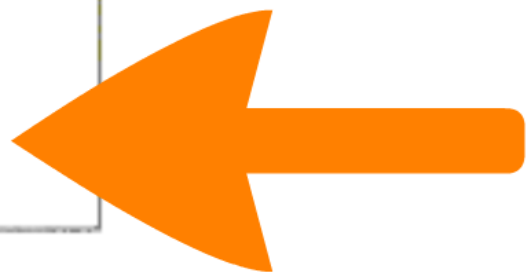
[*13·12] $\equiv . \alpha \cap \beta = \Lambda$ (1)

$\vdash . (1) . *11·11·35 . \supset$

$\vdash :. (\exists x, y) . \alpha = \iota'x . \beta = \iota'y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda$ (2)

$\vdash . (2) . *11·54 . *52·1 . \supset \vdash . \text{Prop}$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.



Really overkill for most systems. However, it is used commercially, e.g.:
seL4 - Formally verified OS microkernel
Various cryptographic protocols
Various embedded software

Much more common in hardware systems, though, where I/O is very well specified and errors can be catastrophic

Why is it overkill?

$$pi(x) > li(x)$$

It's hard.

There are so many special cases.

Property-Based Testing and Formal Verification