CS1632, LECTURE 18: STATIC ANALYSIS, PART 1

Bill Laboon

Static vs Dynamic Analysis

- Static analysis – Code is not executed by the test
- Dynamic analysis – Code is executed by the test.
  - *Almost everything that we have done so far!*

Kinds of Static Analysis

- Code coverage
- Code review / walk-through
- Code metrics
- Formal verification
- Compilers (technically!)
- Bug finders
- Linters

Code Coverage

- How much of the codebase is covered by a particular test suite.
- This can have different meanings!

Code Coverage

Consider the following code and tests and (pseudocode) unit tests

```
class Duck {
   public String quack(int x) {
      if (x > 0) {
         return "Quack!";
      } else {
         return “Negative Quack!”;
      }
   }
   public String quock() {
    return "Quock!";
   }
}


assertEquals(quack(1), “Quack!”);
assertEquals(quack(-4), “Negative Quack!”);
```

Method Coverage

- What percentage of all methods have been called?
- In previous example, 50%

Code Coverage

- Consider the following code and tests and (pseudocode) unit tests

```
public static int noogie(int x) {
   if (x < 10) {
      return 1;
   } else {
      if ((int) Math.sqrt(x) % 2 == 0) {
         return (x / 0);
      } else {
         return 3;
      }
   }
}
```

```
assertEquals(noogie(5), 1);
assertEquals(noogie(81), 3);
assertEquals(noogie(9), 3);
```

Statement Coverage

- That’s 100% method coverage, but we are missing some statements!
- Statement coverage = percentage of statements that have been tested
- This is usually what’s referred to as “code coverage” (although technically it’s a *kind* of code coverage)

Other Kinds of Code Coverage

- Branch coverage
- Condition coverage
- Decision coverage
- Parameter value coverage
- JJ-path Coverage
- Path Coverage
- Entry/Exit Coverage
- State coverage

What’s the benefit?

Code coverage metrics lets you easily see where more tests would be useful and where tests are missing.

It does NOT tell you whether your tests are valid, or whether that line will always work, only that it will be executed by a test!

Consider the following…

```
public int chirp(int x, int y) {
    double z = Math.sqrt(x);
    return (int) z / y;
}

// 100% code coverage!  WOO-HOO!

assertEquals(chirp(1, 1), 1);
```

Note

- Low code coverage is bad, but high code coverage does not always mean good.
- Even 100% of code coverage cannot catch 100% of bugs!

Things Code Coverage Can’t Catch

- Different variable values
- Performance issues
- Race conditions
- Permutations of a statement
- Combinatoric issues
- Anything subjective
- More!

Code Metrics

- Allow you to check :
  - *Cyclomatic complexity!*
  - *Class fan-out!*
  - *Number of lines per class!*
  - *Number of interfaces!*
  - *Depth of inheritance tree!*
  - *NORM (Number of overridden methods)*
  - *Weighted methods per class!*
  - *Afferent and Efferent Coupling!*

Code Metrics

Honestly, I have not found most of these very useful (except cyclomatic complexity, occasionally).

Some people/companies swear by them, though.

You can set “triggers” for these in checkstyle.

Formal Verification

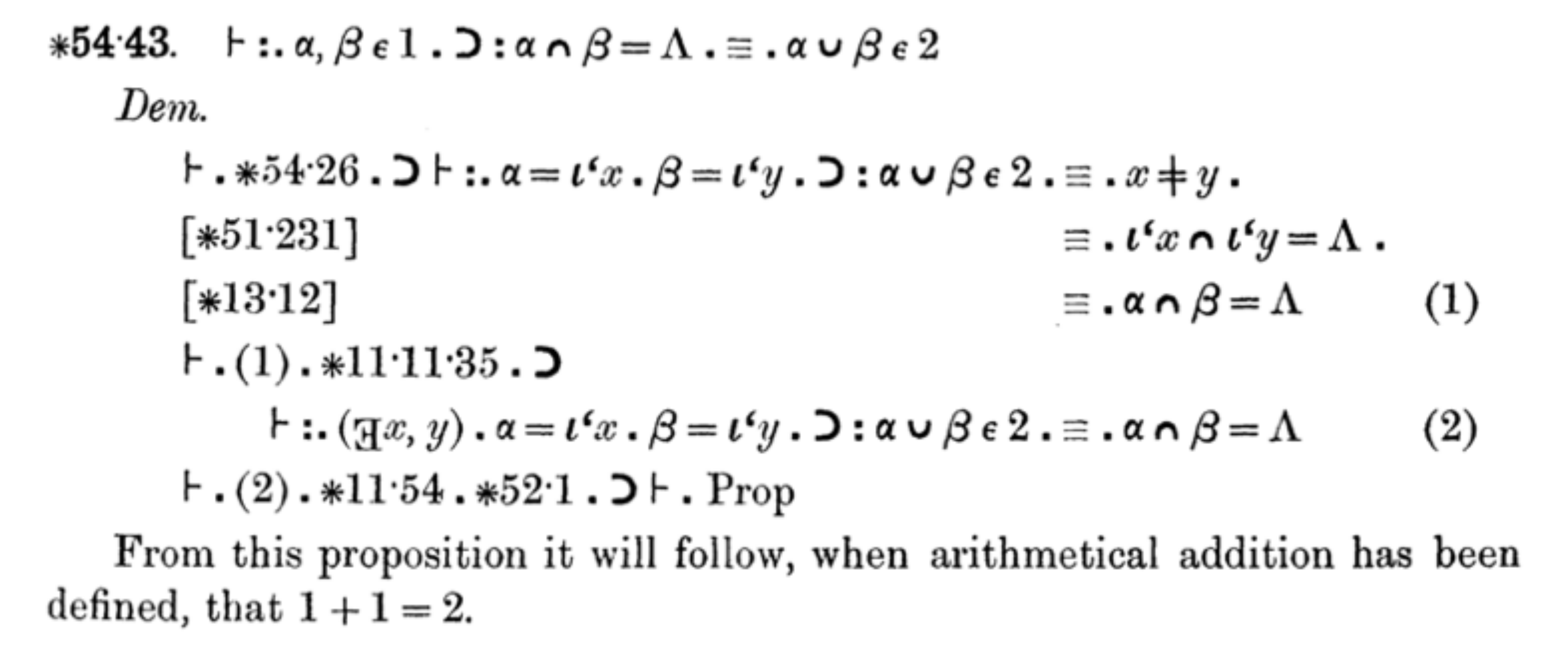
- Mathematically prove the behavior of a program from first principles
- Allows for (in principle) defect-free code
- Often done by reducing to a non-Turing-complete language to avoid the Halting Problem

Wait, did you say defect-free code?!?!

- Sure did. Go back and see.

So why aren’t we using formal verification all the time?

Page 379 of the Principia Mathematica



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

Dem.

$$\vdash . \ast 54\cdot 26 . \supset \vdash :. \alpha = \iota\text{‘}x . \beta = \iota\text{‘}y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$$

$$[\ast 51\cdot 231] \qquad \equiv . \iota\text{‘}x \cap \iota\text{‘}y = \Lambda .$$

$$[\ast 13\cdot 12] \qquad \equiv . \alpha \cap \beta = \Lambda \qquad (1)$$

$$\vdash . (1) . \ast 11\cdot 11\cdot 35 . \supset$$

$$\vdash :. (\exists x, y) . \alpha = \iota\text{‘}x . \beta = \iota\text{‘}y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda \qquad (2)$$

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

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

Formal Verification Is Slow and Inflexible

- Used by:
  - *Some embedded programs*
  - *Some cryptography libraries*
  - *At least one kernel, seL4*
  - *Places where accuracy is absolutely paramount*

And assumes you formally verified correctly!

- You may have a bug in your formal verification software
- You may not have specified it correctly - (remember, verification is not the same as validation)
- Does not take any subjective factors into account (e.g. usability of program, scalability, etc.)

Compilers! Compilers! Compilers!

A compiler can tell you…

- If your syntax is right
- If you have uncaught exceptions
- If you have dead code
- The javac compiler isn’t even the best one out there, but it can give you lots of information
- See the rustc and elm compilers for some really good static analysis