CPEN 422
Static Analysis Techniques and Tools

Quinn Hanam

Static vs. Dynamic Analysis

Simple:

Dynamic analyses execute the program.

Static analyses inspect the source code.

Why should you care?

1. Finds lots of bugs!
2. Scales to MLOC.
3. Better coverage than dynamic analysis.
4. Allows reasoning about programs that are too complex for humans.
5. Can help make you a better programmer.

Some Basic Types of Static Analysis

Parsers

Points-to analysis

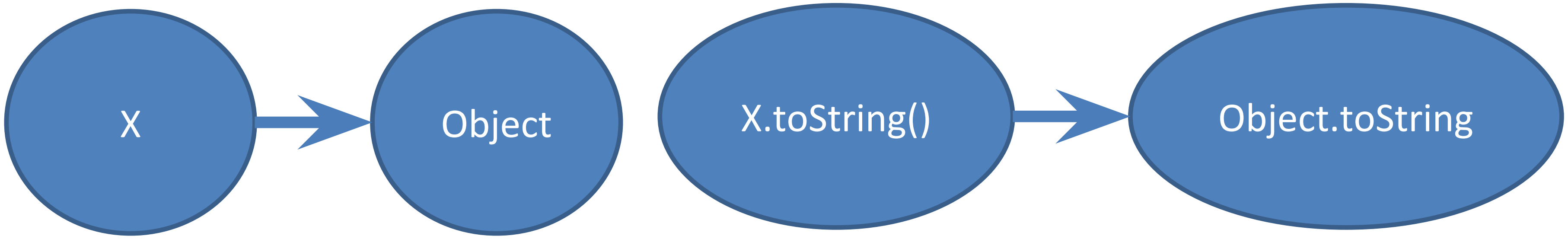
Control flow analysis

Data flow analysis

Basic Types of Static Analysis

Points-to analysis

```
void main(String[] args) {
    Object x = new Object();
    x.toString();
}
```

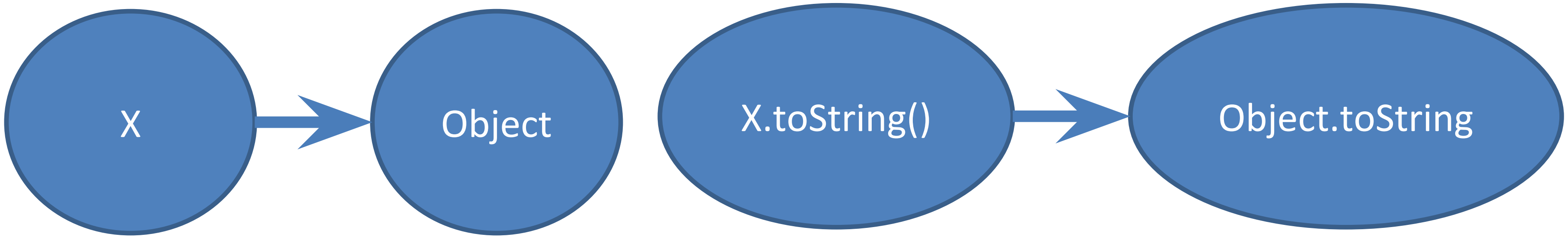


Basic Types of Static Analysis

Points-to analysis

For statically typed languages, we only need the class hierarchy.

Useful for IDE support (e.g., code completion)

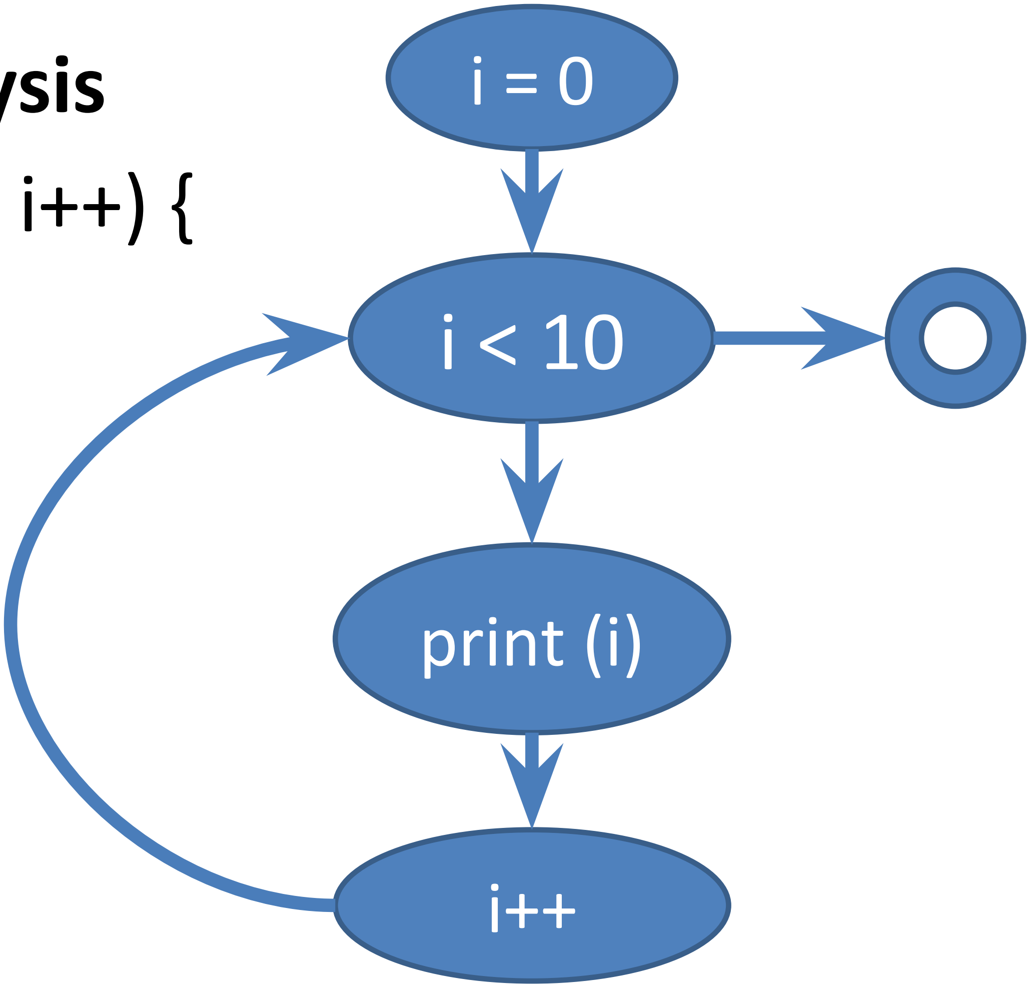


Basic Types of Static Analysis

Control Flow Analysis

```
for( int i = 0; i < 10; i++) {
    print( i );
}
```

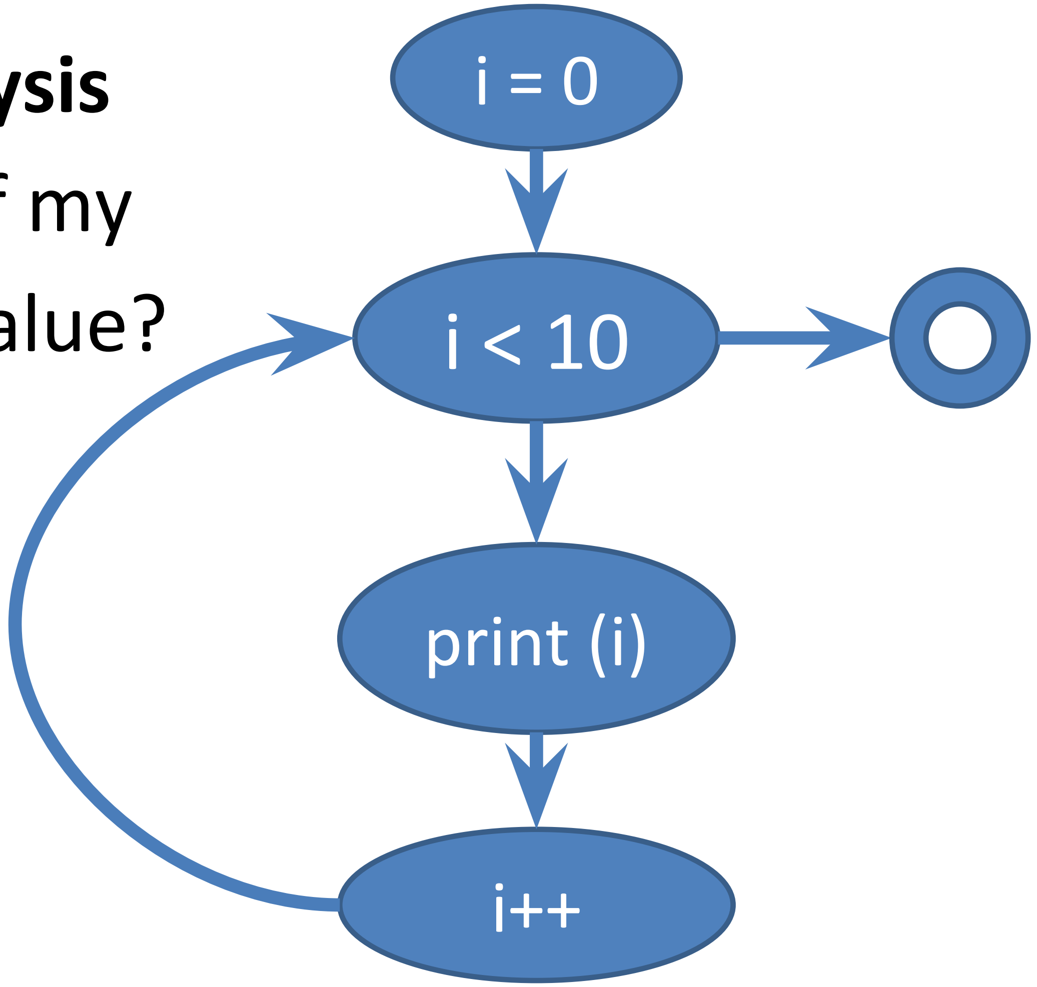
Walk through the control flow graph.



Basic Types of Static Analysis

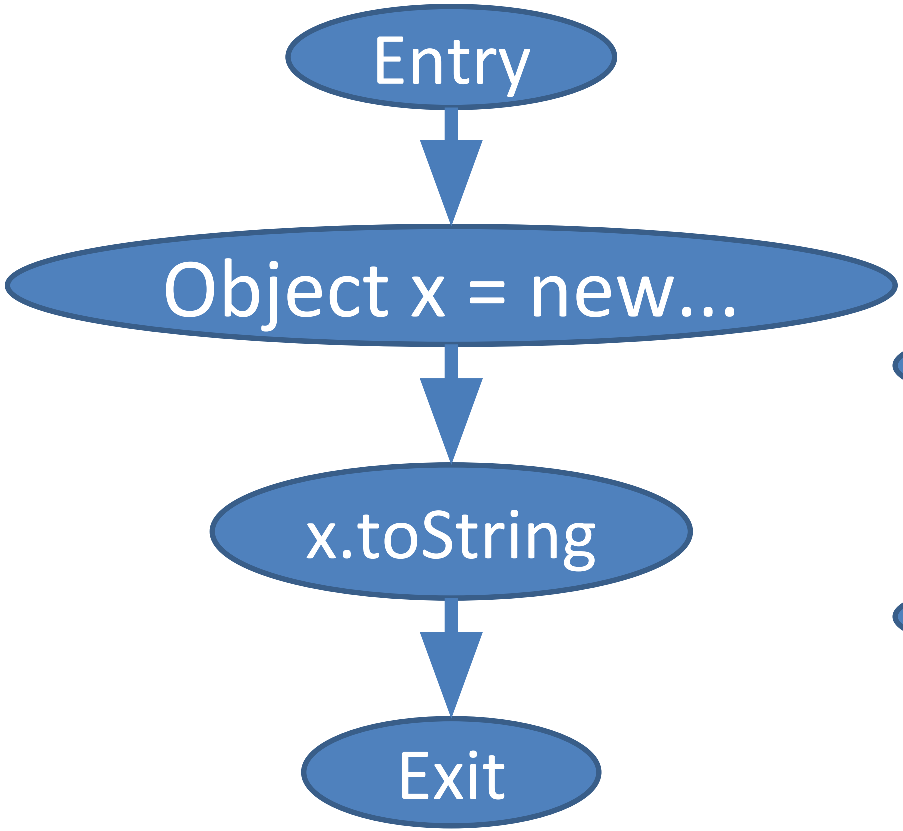
Control Flow Analysis

e.g., do all paths of my method return a value?

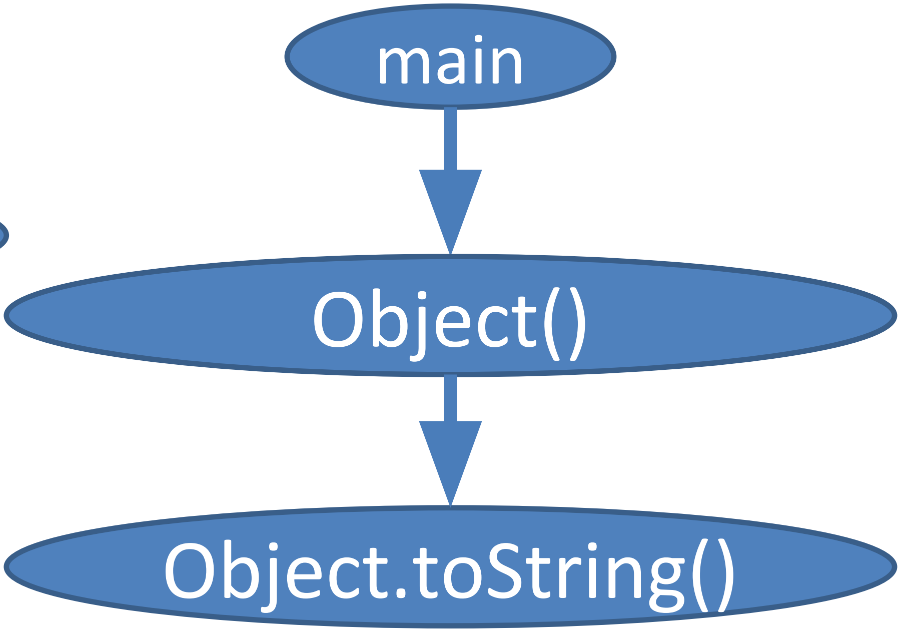


```
void main(String[] args) {
   Object x = new Object();
   x.toString();
}
```

Intra-procedural analysis



Inter-procedural analysis



Basic Types of Static Analysis

Data Flow Analysis

Integer **a**( ) { return new Integer(5); }	Integer **b**() { return null; }

void **main**( ) {
    printValue(a());
    printValue(b());
}
	void **printValue**(Integer v) {
	    System.out.println(v);
	}

	Symbolic value A	**Symbolic value B**
return new integer(5);	5	
return null;		null
printValue();	5	null
System.out.println(v);	5	null

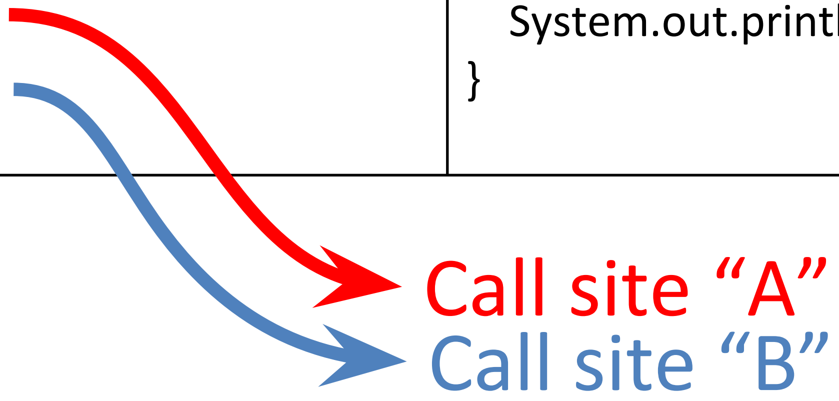
Basic Types of Static Analysis

Data flow analysis is hard!

Performing a **context-sensitive** analysis leads to **path explosion**.

`Integer a( ) { return new Integer( 5 );  }`	`Integer b() { return null; }`

`void main( ) {`
`   printValue( a() );`
`   printValue( b() );`
`}`
	`void printValue( Integer v ) {`
	`   System.out.println( v );`
	`}`



Inferring Specifications

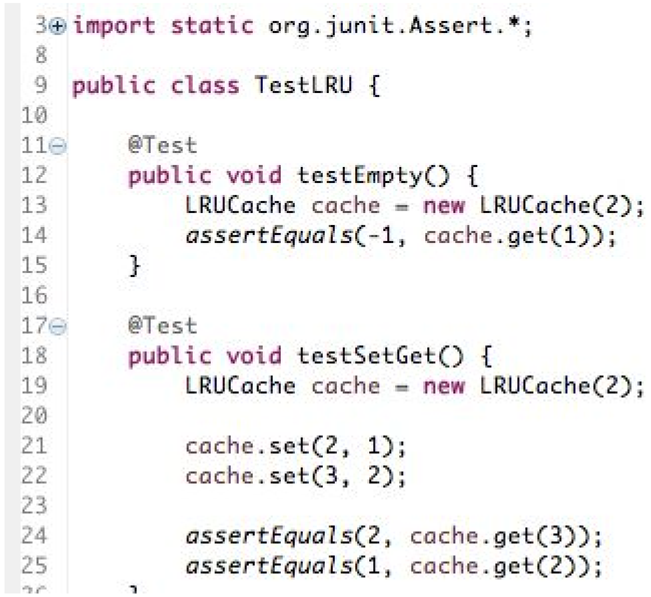
All bug finding tools need specifications in order to work.

I.e. we need to know the correct behavior in order to test that the program behaves correctly.

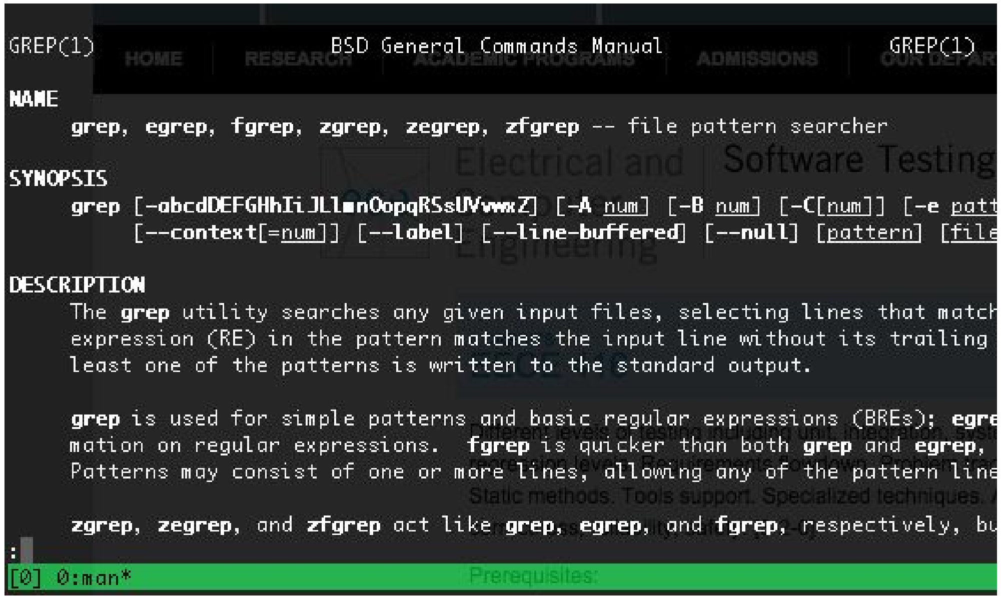
Where can we get specifications from?

Inferring Specifications

Where can we get specifications from?



```java
import static org.junit.Assert.*;

public class TestLRU {

    @Test
    public void testEmpty() {
        LRUCache cache = new LRUCache(2);
        assertEquals(-1, cache.get(1));
    }

    @Test
    public void testSetGet() {
        LRUCache cache = new LRUCache(2);

        cache.set(2, 1);
        cache.set(3, 2);

        assertEquals(2, cache.get(3));
        assertEquals(1, cache.get(2));
```



```
GREP(1)                  BSD General Commands Manual                  GREP(1)

NAME
     grep, egrep, fgrep, zgrep, zegrep, zfgrep -- file pattern searcher

SYNOPSIS
     grep [-abcdDEFGHhIiJLlmnOopqRSsUVwxZ] [-A num] [-B num] [-C[num]] [-e patt
          [--context[=num]] [--label] [--line-buffered] [--null] [pattern] [file

DESCRIPTION
     The grep utility searches any given input files, selecting lines that match
     expression (RE) in the pattern matches the input line without its trailing
     least one of the patterns is written to the standard output.

     grep is used for simple patterns and basic regular expressions (BREs); egre
     mation on regular expressions.  fgrep is quicker than both grep and egrep,
     Patterns may consist of one or more lines, allowing any of the pattern line

     zgrep, zegrep, and zfgrep act like grep, egrep, and fgrep, respectively, bu
:
[0] 0:man*
```

Inferring Specifications

Where can we get specifications from?

```
public class LRUList{
    private Node mru;
    private Node lru;

    /* Inserts a node into the list as the MRU node. */
    public void insert(Node node){
        if(this.mru == null){
            this.mru = node;
            this.lru = node;
        }
        else{
            node.next = mru;
            mru.previous = node;
            this.mru = node;
        }
        assert(this.mru == node);
    }
```

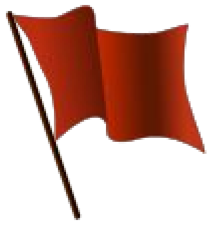
Inferring Specifications

Specifications can come from:

- Test cases (expected output)
- Formal
- Assert() statements
- Source code
- Code comments (iComment)
- Man pages (DASE)
- Forums (e.g., Stack Overflow)
- …

Programmer Beliefs

We infer program specifications from source code using “beliefs”.



Must beliefs (must be true):

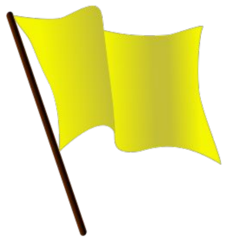
Program	Property inferred
`x = *p / z;`	p != null
	z != 0
`unlock(l);`	l acquired
`x++;`	x not protected by l

We only need one contradiction to infer a bug.

Programmer Beliefs

We infer program specifications from source code using “beliefs”.

May beliefs (may be true):



- We do a statistical analysis to infer a bug

[Adapted from Tan, 09-Beliefs-iComment - February 1, 2015]

Programmer Beliefs

May beliefs (may be true):

- We do a statistical analysis to infer a bug

```
functX(){
  A();
  B();
}
```

```
functY(){
  A();
  B();
}
```

```
functZ(){
  A();
}
```

Support for {A, B} = # occurrences = 2

Support for {A} = 3

Confidence for {A, B} = support({A,B}) / support({A})
= 2/3

General Purpose Static Analysis

Example from Apache Tomcat6

```
if((null == o && null == n.value())) return;
o.value();
```

General Purpose Static Analysis

Example from Apache Tomcat6

```
if((null == o && null == n.value())) return;
o.value();
```

Beliefs:

null == o	o could be null
null == n.value	n.Value() could return null
o.value()	o is not null

General Purpose Static Analysis

Example from Apache Tomcat6

```
1   If((null == o && null == n.value())) return;
2   o.value();
```

Possible values of o at line 2 include null!

Alert: There is a branch that guarantees a null dereference is executed.

Activity

Get into groups of two or three.

Develop your own static analysis “checker”:

1. What defect are you checking for?
2. What properties do you need to infer?
3. What kind of analysis will detect the defect?

You have ~10 minutes

Activity Example

1. What defect are you checking for?

   Null pointer exception

1. What properties do you need to infer?

   Variable being dereferenced could have value of 'null' on some path before it is dereferenced.

2. What kind of analysis will detect the defect?

   Data flow analysis

Checker Examples

Must:

- NPE
- Lock not acquired
- Resource not closed
- Input not sanitized

May:

- Equals() method not overloaded by subclass
- Variable not protected by lock
- Lock inversion
- Call super

Static Analysis Implementations

Compilers

- Syntax
- Optimizations
- Undeclared vars
- Missing return statements
- Uncaught exceptions
- …

Static Analysis Implementations

‘Linting’ Tools

- Define a ‘professional’ subset of the language.
- Check for violations
- Similar to compile time checkers, but check for bad practice
- E.g.:
  - Lint
  - JSLint

[Based on Douglas Crockford’s “http://www.jslint.com/lint.html”]

Static Analysis Implementations

IDE Support Tools

- Type inference
- Code completion

Static Analysis Bug Finding Platforms

- Java
  - **FindBugs**
- C/C++
  - **Clang**
  - CppCheck
- Commercial
  - Coverity
  - CodeSonar

FindBugs (and Clang SA)

The Goal: Find easy-to-detect bugs and bad practice.

The Solution: A platform for “bug detectors”.

Different checkers need different levels of analysis:

- Class structure/inheritance
- Linear code scan
- Control flow analysis
- Data flow analysis

[Hovemeyer and Pugh, “Finding Bugs is Easy”, 2004]

FindBugs (and Clang SA)

Is it useful?

Yes! See “The Google FindBugs Fixit” (Ayewah and Pugh).

[Hovemeyer and Pugh, “Finding Bugs is Easy”, 2004]

Klee Symbolic Execution

Symbolically executes a program under test.

Generates test cases when:

1. The program terminates (obtain high coverage test cases).
2. An error is found

Klee uses a constraint solver to generate inputs based on the symbolic values at termination/error.

[Cadar, Dunbar and Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”, 2004]

Research in Static Analysis @UBC

Static Analysis of JavaScript

Statically analyzing JavaScript is hard:

- Higher order functions
- Dynamically typed
- Fields dynamically assigned and accessed
- Working with c/c++ libraries to access DOM
- Program dynamically built from server side scripts (e.g., PHP)
- Eval()

Commit Changes

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

```
if (isSpecialDeal()) {
    total = price * 0.95;
}
else {
    total = price * 0.98;
}

send();
```

New feature, bug fix or refactoring?

Refactoring

Consolidate duplicate conditional fragments

Commit Changes

```
host = this.get('Host');
return host.split(':')[0];
```

```
host = this.get('Host');
if (!host) return;
return host.split(':')[0];
```

New feature, bug fix or refactoring?

Bug fix

Bug: Dereferenced Non-value

Repair: Protect with falsey check

Summary

- Lots of ways to statically analyze a program, e. g.: points-to, control flow and data flow analysis.
- We can infer program specifications from source code.
- We can find bugs using static analysis by finding contradictions to those specifications.
- There are many static analysis tools out there, and you should use them!

Further Reading

- “Bugs as Deviant Behavior” (Engler et. al.)
- “Finding Bugs is Easy!” (Hovermeyer and Pugh)
- “KLEE: Unassisted and Automatic Generation of High Coverage Tests for Complex Systems Programs”(Cadar, Dunbar and Engler)
- “Finding Patterns in Static Analysis Alerts” (Hanam et. al.)
- “iComment”(Tan et. al.)
- “DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing” (Wong/Zhang et. al.)