Carnegie Mellon University

Lecture 2: End of Syntactic Analysis, start of Program Semantics

17-355/17-665/17-819: Program Analysis

Claire Le Goues and Fraser Brown

Jan 19, 2023

* Course materials developed with Jonathan Aldrich and Rohan Padyhe

S3D Software and Societal Systems Department

Administrivia

- HW1 out; due next week.
  - Lots of references online, many linked in assignment.
  - Recitation will have some practice problems
  - Submit via Canvas/Gradescope. Copy/paste your queries + screenshots of analysis results.
- Office hours are up on website.
- Make sure you're on Piazza.

Learning objectives

- Describe the function of an AST and outline the principles behind AST walkers and declarative languages for simple bug-finding analyses.
- Recognize the basic WHILE demonstration language and translate between WHILE and While3Addr.
- Define the meaning of programs using operational semantics
- Read and write inference rules and derivation trees
- Use big- and small-step semantics to show how WHILE programs evaluate
- Use structural induction to prove things about program semantics

Remember this code?

```
void copy_bytes(char dest[], char source[], int n) {
    for (int i = 0; i < n; ++i)
        dest[i] = source[i];
}
```

```
void copy_bytes(char dest[], char source[], int n) {
    for (int i = 0; i < n; ++i)
        dest[i] = source[i];
}
```

Here’s the AST Clang makes:

```
|-FunctionDecl 0x13900a3d8 <foo.c:1:1, line:4:1> line:1:6 copy_bytes 'void (char *, char *, int)'
| |-ParmVarDecl 0x13900a1b0 <col:17, col:27> col:22 used dest 'char *':'char *'
| |-ParmVarDecl 0x13900a238 <col:30, col:42> col:35 used source 'char *':'char *'
| |-ParmVarDecl 0x13900a2b8 <col:45, col:49> col:49 used n 'int'
| `-CompoundStmt 0x13900a7e8 <col:52, line:4:1>
|   `-ForStmt 0x13900a7b0 <line:2:3, line:3:23>
|     |-DeclStmt 0x13900a578 <line:2:8, col:17>
|     | `-VarDecl 0x13900a4f0 <col:8, col:16> col:12 used i 'int' cinit
|     |   `-IntegerLiteral 0x13900a558 <col:16> 'int' 0
|     |-<<<NULL>>>
|     |-BinaryOperator 0x13900a600 <col:19, col:23> 'int' '<'
|     | |-ImplicitCastExpr 0x13900a5d0 <col:19> 'int' <LValueToRValue>
|     | | `-DeclRefExpr 0x13900a590 <col:19> 'int' lvalue Var 0x13900a4f0 'i' 'int'
|     | `-ImplicitCastExpr 0x13900a5e8 <col:23> 'int' <LValueToRValue>
|     |   `-DeclRefExpr 0x13900a5b0 <col:23> 'int' lvalue ParmVar 0x13900a2b8 'n' 'int'
|     |-UnaryOperator 0x13900a640 <col:26, col:28> 'int' prefix '++'
|     | `-DeclRefExpr 0x13900a620 <col:28> 'int' lvalue Var 0x13900a4f0 'i' 'int'
|     `-BinaryOperator 0x13900a790 <line:3:5, col:23> 'char' '='
|       |-ArraySubscriptExpr 0x13900a6c8 <col:5, col:11> 'char' lvalue
|       | |-ImplicitCastExpr 0x13900a698 <col:5> 'char *':'char *' <LValueToRValue>
|       | | `-DeclRefExpr 0x13900a658 <col:5> 'char *':'char *' lvalue ParmVar 0x13900a1b0 'dest' 'char *':'char *'
|       | `-ImplicitCastExpr 0x13900a6b0 <col:10> 'int' <LValueToRValue>
|       |   `-DeclRefExpr 0x13900a678 <col:10> 'int' lvalue Var 0x13900a4f0 'i' 'int'
|       `-ImplicitCastExpr 0x13900a778 <col:15, col:23> 'char' <LValueToRValue>
|         `-ArraySubscriptExpr 0x13900a758 <col:15, col:23> 'char' lvalue
|           |-ImplicitCastExpr 0x13900a728 <col:15> 'char *':'char *' <LValueToRValue>
|           | `-DeclRefExpr 0x13900a6e8 <col:15> 'char *':'char *' lvalue ParmVar 0x13900a238 'source' 'char *':'char *'
|           `-ImplicitCastExpr 0x13900a740 <col:22> 'int' <LValueToRValue>
|             `-DeclRefExpr 0x13900a708 <col:22> 'int' lvalue Var 0x13900a4f0 'i' 'int'
```

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for the pattern in question.
- Various frameworks, some more language-specific than others.
  - Tradeoffs between language agnosticism and semantic information available.
  - Consider “grep”: very language agnostic, not very smart.
  - Python’s ”astor” package designed for Python ASTs. Clean API; highly specific.
- Classic architecture based on Visitor pattern:
  - class Visitor has a visitX method for each type of AST node X
  - Default Visitor code just descends the AST, visiting each node
  - To do something interesting for AST element of type X, override visitX
- **More recent approaches based on semantic search, declarative logic programming, or query languages.**

CodeQL

- A language for querying code. Developed by GitHub.
- Supports many common languages.
- Library of common programming patterns and optimizations.

Inefficient empty string test

```
ID: java/inefficient-empty-string-test
Kind: problem
Severity: recommendation
Precision: high
Tags:
   - efficiency
   - maintainability
Query suites:
   - java-security-and-quality.qls
```

Click to see the query in the CodeQL repository

When checking whether a string `s` is empty, perhaps the most obvious solution is to write something like `s.equals("")` (or `"".equals(s)`). However, this actually carries a fairly significant overhead, because `String.equals` performs a number of type tests and conversions before starting to compare the content of the strings.

Recommendation

The preferred way of checking whether a string `s` is empty is to check if its length is equal to zero. Thus, the condition is `s.length() == 0`. The `length` method is implemented as a simple field access, and so should be noticeably faster than calling `equals`.

Note that in Java 6 and later, the `String` class has an `isEmpty` method that checks whether a string is empty. If the codebase does not need to support Java 5, it may be better to use that method instead.

Example

In the following example, class `InefficientDBClient` uses `equals` to test whether the strings `user` and `pw` are empty. Note that the test `"".equals(pw)` guards against `NullPointerException`, but the test `user.equals("")` throws a `NullPointerException` if `user` is `null`.

In contrast, the class `EfficientDBClient` uses `length` instead of `equals`. The class preserves the behavior of `InefficientDBClient` by guarding `pw.length() == 0` but not `user.length() == 0` with an explicit test for `null`. Whether or not this guard is desirable depends on the intended behavior of the program.

```
// Inefficient version
class InefficientDBClient {
        public void connect(String user, String pw) {
                if (user.equals("") || "".equals(pw))
                        throw new RuntimeException();
                ...
        }
}
```

https://codeql.github.com/codeql-query-help/java/java-inefficient-empty-string-test/

S3D

Example: Java string compare with “”

```java
// Inefficient version
class InefficientDBClient {
    public void connect(String user, String pw) {
        if (user.equals("") || "".equals(pw))
            throw new RuntimeException();
        ...
    }
}

// More efficient version
class EfficientDBClient {
    public void connect(String user, String pw) {
        if (user.length() == 0 || (pw != null && pw.length() == 0))
            throw new RuntimeException();
        ...
    }
}
```

Hint: doub

CodeQL query for empty string comparison

Query: InefficientEmptyStringTest.ql ⌄ Collapse source

```
/**
 * @name Inefficient empty string test
 * @description Checking a string for equality with an empty string is inefficient.
 * @kind problem
 * @problem.severity recommendation
 * @precision high
 * @id java/inefficient-empty-string-test
 * @tags efficiency
 *       maintainability
 */

import java

from MethodAccess mc
where
  mc.getQualifier().getType() instanceof TypeString and
  mc.getMethod().hasName("equals") and
  (
    mc.getArgument(0).(StringLiteral).getRepresentedString() = "" or
    mc.getQualifier().(StringLiteral).getRepresentedString() = ""
  )
select mc, "Inefficient comparison to empty string, check for zero length instead."
```

https://github.com/github/codeql/blob/main/java/ql/src/Performance/InefficientEmptyStringTest.ql

S3D

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for the pattern in question.
- Various frameworks, some more language-specific than others.
  - Tradeoffs between language agnosticism and semantic information available.
  - Consider “grep”: very language agnostic, not very smart.
  - Python’s ”astor” package designed for Python ASTs. Clean API; highly specific.
- **Classic architecture based on Visitor pattern:**
  - class Visitor has a visitX method for each type of AST node X
  - Default Visitor code just descends the AST, visiting each node
  - To do something interesting for AST element of type X, override visitX
- More recent approaches based on semantic search, declarative logic programming, or query languages.

Together: String concatenation in a loop

- Pseudocode for a simple syntactic analysis that warns **when string concatenation occurs in a loop**
  - Why? In Java and .NET it may be more efficient to use a StringBuffer

```
class StringConcatLoopAnalysis extends Visitor {
  private int loopLevel = 0;

  void visitStringConcat(StringConcat e) {
    if (loopLevel > 0)
      warn("Use StringBuffer instead)")
    super.visitStringConcat(e); // visits children
  }

  void visitWhile(While e) {
    loopLevel++;
    super.visitWhile(e);  // visits children
    loopLevel--;
  }

 // similar for other looping constructs
```