Static Analysis

Dr. Sebastian Proksch

Advanced Software Engineering
Spring 2018
Definition

Static program analysis is the **analysis** of computer **software** that is performed **without** actually **executing** programs.

https://en.wikipedia.org/wiki/Static_program_analysis
All Representations Can Be Analyzed

Focus of this lecture is analyzing source code.
Goals

Quality Assurance

Optimization Opportunities

Detect Vulnerabilities

→ Show presence or absence of a specific property.
Examples

• Are all enum cases handled in a switch?
• Does equals/hashcode consider all fields of a class?
  • What about super fields?
  • What about the super implementation?
  • Are fields only accessed or actually included in value?
• Does the program contain dead code?
• Program slicing (i.e., “Show me the part of the function that is related to statement X”)?
• ..?
Recommended Reading

Principles of Program Analysis
Soundness
Terminology

• **Soundness**
  • A system is sound if properties it can prove are indeed true in the model

• **Completeness**
  • A complete system can prove any true property of a model

→ Static analysis on real applications cannot be both sound and complete!

https://en.wikipedia.org/wiki/G%C3%B6del%27s_incompleteness_theorems
Precision and Recall

https://en.wikipedia.org/wiki/Precision_and_recall
Three “Invalid Program” Detectors

Tool A is *sound*, it never approves an invalid program (low recall, perfect precision)

Tool B is *unsound* (higher recall, non-perfect precision)

Tool C is *complete*, but *unsound* (perfect recall)

For Tool B:
- p1: True Positive
- p2: False Negative
- p3: False Positive
- p4: True Negative
The WHILE Language
The WHILE Language

Syntactic Categories:
\( a \) arithmetic expression
\( b \) boolean expression
\( S \) statement

Simple Name Conventions:
\( x, y, \ldots \) variables
\( n \) numerals
\( L \) labels

Operators:
\( op_a \) arithmetic operators
\( op_b \) Boolean operators
\( op_r \) relational operators
The WHILE Abstract Syntax

\[ a := x \mid n \mid a_1 \text{ op}_a a_2 \]

\[ b := \text{true} \mid \text{false} \mid \text{not } b \]
\[ \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \]

\[ S := [x=a]^L \mid [\text{skip}]^L \mid S_1;S_2 \]
\[ \mid \text{if } [b]^L \{ S_1 \} \text{ else } \{ S_2 \} \]
\[ \mid \text{while } [b]^L \{ S \} \]
A WHILE Program: FACT

\[
\begin{align*}
[y &= x] & \quad ^1; \\
[z &= 1] & \quad ^2; \\
\textbf{while} & [y > 1] \quad ^3 \{ \\
[&z = z \times y] & \quad ^4; \\
[&y = y - 1] & \quad ^5; \\
\} \\
[y &= 0] & \quad ^6;
\end{align*}
\]
Reaching Definition Analysis

An assignment of the form $[x = a]^L$ may reach a certain program point if there is an execution of the program where $x$ was last assigned a value at $L$ when the program point is reached.
# Reaching Definitions for FACT

<table>
<thead>
<tr>
<th>L</th>
<th>( \text{RD}_{\text{entry}}(L) )</th>
<th>( \text{RD}_{\text{exit}}(L) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( x:?, y:?, z:? )</td>
<td>( x:?, y:1, z:? )</td>
</tr>
<tr>
<td>2</td>
<td>( x:?, y:1, z:? )</td>
<td>( x:?, y:1, z:2 )</td>
</tr>
<tr>
<td>3</td>
<td>( x:?, y:{1,5}, z:{2,4} )</td>
<td>( x:?, y:{1,5}, z:{2,4} )</td>
</tr>
<tr>
<td>4</td>
<td>( x:?, y:{1,5}, z:{2,4} )</td>
<td>( x:?, y:{1,5}, z:4 )</td>
</tr>
<tr>
<td>5</td>
<td>( x:?, y:{1,5}, z:4 )</td>
<td>( x:?, y:5, z:4 )</td>
</tr>
<tr>
<td>6</td>
<td>( x:?, y:{1,5}, z:{2,4} )</td>
<td>( x:?, y:6, z:{2,4} )</td>
</tr>
</tbody>
</table>
Control Flow Graph of FACT

In contrast to the syntax tree, the control flow graph contains all possible transitions in the program.
The FUNC Language
The FUNC Abstract Syntax

Extends WHILE:

\[ a := \ldots \]
\[ | \ \text{fun} \ (a_1, a_2, \ldots) \ \{ \ S; \ \text{return} \ a_n; \ \} \]
\[ | \ a_1 \ (a_2, a_3, \ldots) \]

Example:

[sum = fun (a, b) { [c = a + b]² return c; }]¹

[\[ z = \text{sum}(x, y) \]³]

The language is not type safe (among other severe limitations), but it serves its purpose for now!
A FUNC Program: ID

$id = \text{fun} \ (o) \ { \text{return} \ o; \ }]^1;$
$x = 42^2;$
$x = id(x)^3;$

What are the exit reaching definitions of $x$ at label 3?

<table>
<thead>
<tr>
<th>L</th>
<th>$\text{RD}_{\text{entry}}(L)$</th>
<th>$\text{RD}_{\text{exit}}(L)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>id:?, x:?</td>
<td>id:1, x:?</td>
</tr>
<tr>
<td>2</td>
<td>id:1, x:?</td>
<td>id:1, x:2</td>
</tr>
<tr>
<td>3</td>
<td>id:1, x:2</td>
<td>id:1, ...</td>
</tr>
</tbody>
</table>

Simple table representation is not sufficient anymore!
Call Graph of SUM

Main:

[sum = «Function»]¹
[z = sum(x, y)]²

Fun1:

fun (a, b)³ {
    [c = a + b;]⁴
    return c;
}

Analysis has to maintain a “call stack” and remember reaching definitions. On call, the actual parameters have to be mapped to the formal parameters. The reaching definition of the return value can be mapped back.

RD_{exit}(2)? z:2? z:4? Depends on the “analysis scope”.

http://matt.might.net/articles/implementation-of-kcfa-and-0cfa/
Analysis Scope

MainClass

Class1

Class2

main

fun1

fun2

fun3

Intra-Procedural Analysis

Intra-Class Analysis

Inter-Procedural Analysis
Static analyses of object-oriented programs must establish upper and lower bounds in the possible variable types to reduce the “fan out” by pruning impossible calls.
Sensitivity
Flow Sensitivity

Which methods were “invoked before”?

Flow Insensitive:
(Global picture on the analysis unit)

Flow Sensitive:
(Order of statements)

Path Sensitive:
(Actual execution path)

Context Sensitive:
(Inter-procedural analysis)
Other Sensitivities

• **Field Sensitivity**
  • Are fields considered independently?
  • “o.f = «tainted value»” ... is o tainted?

• **Loop Sensitivity**
  • How many iterations are considered?

• **Object/Array Sensitivity**
  • “o[1] = «tainted value»” ... is o tainted?

Configuration of the sensitivities controls the tradeoff between soundness, precision, and recall. It also has a significant impact on the runtime and memory consumption of a static analysis.
Refresher: Visitor Pattern
https://en.wikipedia.org/wiki/Visitor_pattern
Intent

Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Visitor Pattern (UML)

Notation: UML
Visitor Pattern (Sequence)
Consequences

• Makes adding new operations easy.
• Gathers related operations and separates unrelated ones.
• Adding new elements to the hierarchy is hard.
• No visiting across hierarchy boundaries.
• Visitors can (but don’t have to) accumulate state.
Practical Exercises
Example: File System Visitor

See Course Material...
Example: Counting Method Calls

How many method calls...
• ... exist per class?
• ... exist in a method? Init?
• ... are guarded by an if?
Take Home Messages
You should be able to answer...

- What is “static analysis”? What are its goals?
- Soundness? Precision? Recall?
- What are reaching definitions? How to find them?
- What is a control flow graph?
- What is a call graph?
- What are analysis scopes?
- What properties affect the sensitivity of an analysis?
- How and when to apply the visitor pattern?
- How to use the provided datasets in the ASE project?