Lecture 1: Introduction to Program Analysis

17-355/17-655/17-819: Program Analysis

Claire Le Goues

January 14, 2020

* Course materials developed with Jonathan Aldrich

Learning objectives

- Provide a high level definition of program analysis and give examples of why it is useful.
- Sketch the explanation for why all analyses must approximate.
- Understand the course mechanics, and be motivated to read the syllabus.
- Describe the function of an AST and outline the principles behind AST walkers for simple bug-finding analyses.
- Recognize the basic WHILE demonstration language and translate between WHILE and While3Addr.

What is this course about?

- Program analysis is the systematic examination of a program to determine its properties.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

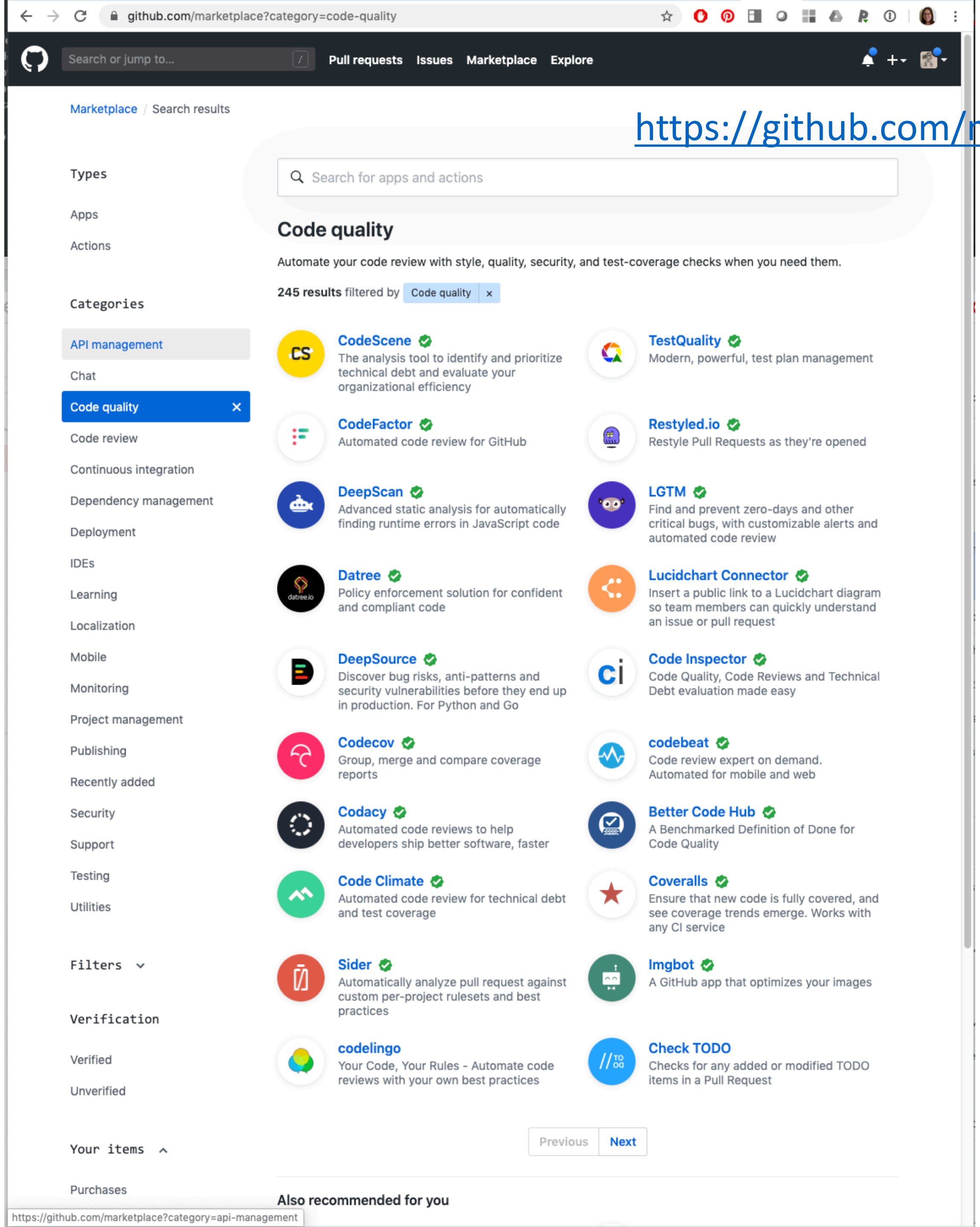
Why might you care?

- Program analysis, and the skills that underlie it, have implications for:
  - Automatic bug finding.
  - Language design and implementation.
  - Program synthesis.
  - Program transformation (refactoring, optimization, repair).

```
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```

```
public int foo() {
    doStuff();

    return 3;

    doMoreStuff();
    return 4;
}
```

https://github.com/marketplace?category=code-quality



```
package com.google.devtools.staticanalysis;

public class Test {
```

Lint — Missing a Javadoc comment.
Java
1:02 AM, Aug 21
Please fix | Not useful

```
  public boolean foo() {
    return getString() == "foo".toString();
```

ErrorProne — String comparison using reference equality instead of value equality (see http://code.google.com/p/error-prone/wiki/StringEquality)
StringEquality
1:03 AM, Aug 21
Please fix
Suggested fix attached: show | Not useful

```
  }

  public String getString() {
    return new String("foo");
  }
}
```

//depot/google3/java/com/google/devtools/staticanalysis/Test.java

`package com.google.devtools.staticanalysis;`	`package com.google.devtools.staticanalysis;`
	`import java.util.Objects;`
`public class Test {`	`public class Test {`
`  public boolean foo() {`	`  public boolean foo() {`
`    return getString() == "foo".toString();`	`    return Objects.equals(getString(), "foo".toString());`
`  }`	`  }`
`  public String getString() {`	`  public String getString() {`
`    return new String("foo");`	`    return new String("foo");`
`  }`	`  }`
`}`	`}`

Apply | Cancel

facebook Engineering

Open Source ⌵ Platforms ⌵ Infrastructure Systems ⌵ Physical Infrastructure ⌵ Video Engineerin

POSTED ON MAY 2, 2018 TO DEVELOPER TOOLS, OPEN SOURCE

Sapienz: Intelligent automated software testing at scale

sapienz

By Ke Mao

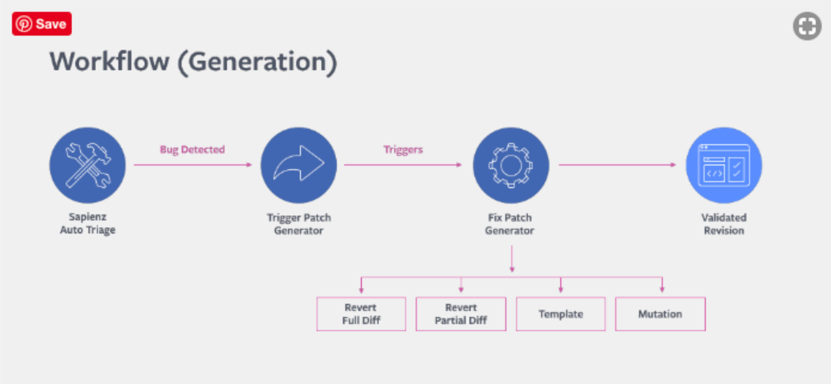
Sapienz technology leverages automated test design to make the testing process faster, more comprehensive, and more effective.

facebook Engineering

Open Source ⌵ Platforms ⌵ Infrastructure Systems ⌵ Physical Infrastructure ⌵ Video Engineering & AR/VR ⌵

POSTED ON SEP 13, 2018 TO AI RESEARCH, DEVELOPER TOOLS, OPEN SOURCE, PRODUCTION ENGINEERING

Finding and fixing software bugs automatically with SapFix and Sapienz



By Yue Jia Ke Mao Mark Harman

Debugging code is drudgery. But SapFix, a new AI hybrid tool created by Facebook engineers, can significantly reduce the amount of time engineers spend on debugging, while also speeding up the process of rolling out new software. SapFix can automatically generate fixes for specific bugs, and then propose them to engineers for approval and deployment to production.

SapFix has been used to accelerate the process of shipping robust, stable code updates to millions of devices using the Facebook Android app — the first such use of AI-powered testing and debugging tools in production at this scale. We intend to share SapFix with the engineering community, as it is the next step in the evolution of automating debugging, with the potential to boost the production and stability of new code for a wide range of companies and research organizations.

SapFix is designed to operate as an independent tool, able to run either with or without Sapienz, Facebook's intelligent automated software testing tool, which was announced at F8 and has already been deployed to production. In its current, proof-of-concept state,

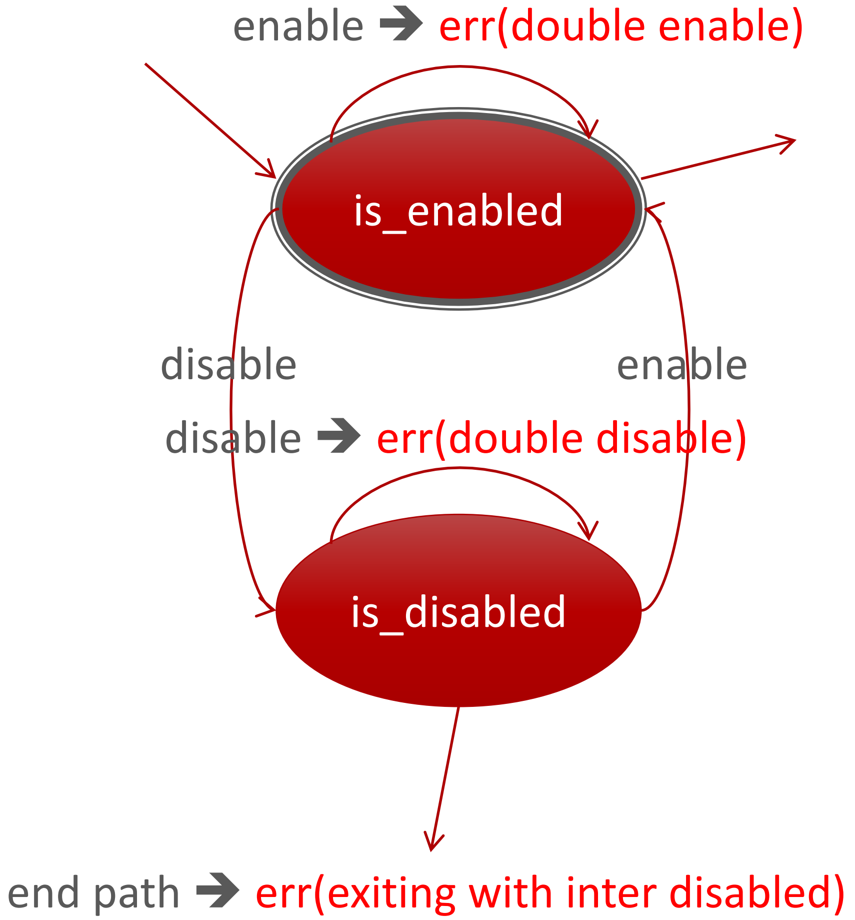
IS THERE A BUG IN THIS CODE?

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
     return NULL;
   sh->buffer_pool = bh -> b_next;
   bh->b_size = b_size;
   restore_flags(flags); // re-enables interrupts
   return bh;
}
```

ERROR: function returns with interrupts disabled!

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
sm check_interrupts {
// variables; used in patterns
decl { unsigned } flags;
// patterns specify enable/disable functions
pat enable = { sti() ; }
               | { restore_flags(flags); } ;
pat disable = { cli() ; }
//states; first state is initial
is_enabled : disable ➔ is_disabled
      | enable ➔ { err("double enable"); }
;
is_disabled : enable ➔ is_enabled
      | disable ➔ { err("double disable"); }
//special pattern that matches when
// end of path is reached in this state
      | $end_of_path$ ➔
          { err("exiting with inter disabled!"); }
;
}
```



Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
     return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Initial state: is_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
     return NULL;
   sh->buffer_pool = bh -> b_next;
   bh->b_size = b_size;
   restore_flags(flags); // re-enables interrupts
   return bh;
}
```

Transition to: is_disabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Final state: is_disabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI ‘000

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
     return NULL;
   sh->buffer_pool = bh -> b_next;
   bh->b_size = b_size;
   restore_flags(flags); // re-enables interrupts
   return bh;
}
```

Transition to: is_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Final state: is_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Behavior of interest…

- Is on uncommon execution paths.
  - Hard to exercise when testing.
- Executing (or analyzing) all paths is infeasible
- **Instead: (abstractly) check the entire possible state space of the program.**

What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

The Bad News: Rice's Theorem

"Any nontrivial property about the language recognized by a Turing machine is undecidable.“

Henry Gordon Rice, 1953

Proof by contradiction (sketch)

Assume that you have a function that can determine if a program *p* has some nontrivial property (like `divides_by_zero`):

```
int silly(program p, input i) {
  p(i);
  return 5/0;
}
bool halts(program p, input i) {
 return divides_by_zero(`silly(p,i)`);
}
```

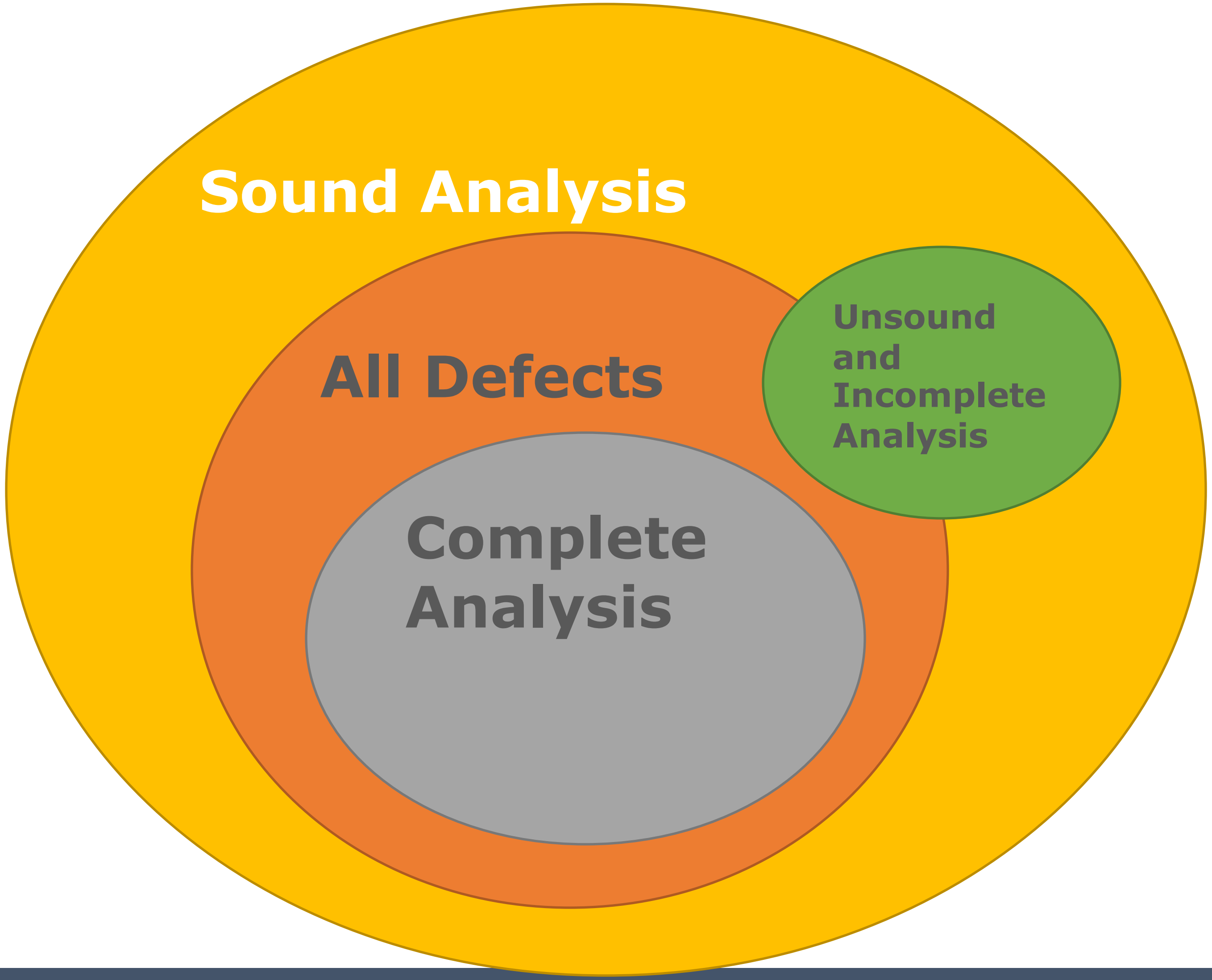
	Error exists	No error exists
Error Reported	True positive (correct analysis result)	False positive
No Error Reported	False negative	True negative (correct analysis result)

Sound Analysis:

reports all defects
-> no false negatives
typically overapproximated

Complete Analysis:

every reported defect is an actual defect
-> no false positives
typically underapproximated


Sound Analysis
All Defects
Unsound and Incomplete Analysis
Complete Analysis

In Defense of Soundiness: A Manifesto

Ben Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Sam Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis

Microsoft Research, Samsung Research America, University of Athens, University of Waterloo, University of Alberta, University of Colorado Boulder, Tufts University, IIT Bombay, Aarhus University, Google

Static program analysis is a key component of many software development tools, including compilers, development environments, and verification tools. Practical applications of static analysis have grown in recent years to include tools by companies such as Coverity, Fortify, GrammaTech, IBM, and others. Analyses are often expected to be *sound* in that their result models all possible executions of the program under analysis. Soundness implies that the analysis computes an over-approximation in order to stay tractable; the analysis result will also model behaviors that do not actually occur in any program execution. The *precision* of an analysis is the degree to which it avoids such spurious results. Users expect analyses to be sound as a matter of course, and desire analyses to be as precise as possible, while being able to *scale* to large programs.

Soundness would seem essential for any kind of static program analysis. Soundness is also widely emphasized in the academic literature. Yet, in practice, soundness is commonly eschewed: we are not aware of a *single* realistic whole-program[1] analysis tool (*e.g.*, tools widely used for bug detection, refactoring assistance, programming automation, *etc.*) that does not purposely make unsound choices. Similarly, virtually all published whole-program analyses are unsound and omit conservative handling of common language features when applied to *real programming languages*.

The typical reasons for such choices are engineering compromises: implementers of such tools are well aware of how they could handle complex language features soundly (*e.g.*, by assuming that a complex language feature can exhibit *any* behavior), but do not do so because this would make the analysis *unscalable* or *imprecise* to the point of being useless. Therefore, the dominant practice is one of treating soundness as an engineering choice.

In all, we are faced with a paradox: on the one hand we have the ubiquity of unsoundness in any practical whole-program analysis tool that has a claim to precision and scalability; on the other, we have a research community that, outside a small group of experts, is oblivious to any unsoundness, let alone its preponderance in practice.

Our observation is that the paradox can be reconciled. The state of the art in realistic analyses exhibits consistent traits, while also integrating a sharp discontinuity. On the one hand, typical realistic analysis implementations have a *sound core*: most common language features are *over-approximated*, modeling all their possible behaviors. Every time there are multiple options (*e.g.*, branches of a conditional statement, multiple data flows) the analysis models all of them. On the other hand, some specific language features, well known to experts in the area, are best *under-approximated*. Effectively, every analysis pretends that perfectly possible behaviors cannot happen. For instance, it is conventional for an otherwise sound static analysis to treat highly-dynamic language constructs, such as Java reflection or *eval* in JavaScript, under-approximately. A practical analysis, therefore, may pretend that *eval* does nothing, unless it can precisely resolve its string argument at compile time.

We introduce the term *soundy* for such analyses. The concept of *soundiness* attempts to capture the balance, prevalent in practice, of over-approximated handling of most language features, yet deliberately under-approximated handling of a feature subset well recognized by experts. Soundiness is in fact what is meant in many papers that claim to describe a sound analysis. A *soundy* analysis aims to be as sound as possible without excessively compromising precision and/or scalability.

Our message here is threefold:

1. We bring forward the ubiquity of, and engineering need for, unsoundness in the static program analysis practice. For static analysis researchers, this may come as no surprise. For the rest of the community, which expects to use analyses as a black box, this unsoundness is less understood.

https://yanniss.github.io/Soundiness-CACM.pdf

[1] We draw a distinction between whole program analyses, which need to model shared data, such as the heap, and modular analyses—e.g., type systems. Although this space is a continuum, the distinction is typically well-understood.

What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- Principal techniques:
  - **Dynamic:**
    - **Testing:** Direct execution of code on test data in a controlled environment.
    - **Analysis:** Tools extracting data from test runs.
  - **Static:**
    - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
    - **Analysis:** Tools reasoning about the program without executing it.
  - ...and their combination.

Course topics

- Program representation
- Abstract interpretation: Use abstraction to reason about possible program behavior.
  - Operational semantics.
  - Dataflow Analysis
  - Termination, complexity
  - Widening, collecting
  - Interprocedural analysis
  - Datalog
  - Control flow analysis
- Hoare-style verification: Make logical arguments about program behavior.
  - Axiomatic semantics
  - Separation logic: modern bug finding.
- Symbolic execution: test all possible executions paths simultaneously.
  - Concolic execution
  - Test generation
- SAT/SMT solvers
- Program synthesis
- Dynamic analysis
- Program repair
- Model checking (briefly) : reason exhaustively about possible program states.
  - Take 15-414 if you want the full treatment!
- We will basically *not* cover types.

Fundamental concepts

- Abstraction.
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- The importance of semantics.
  - We prove things about analyses with respect to the semantics of the underlying language.
- Program proofs as inductive invariants.
- Implementation
  - You do not understand analysis until you have written several.

Course mechanics

When/what.

- Lectures 2x week (T,Th).
  - Mostly *not* using slides (...this first lecture notwithstanding).
  - Instead: board, lecture notes, exercises.
  - Bring a pen/pencil.
  - Try to stay off your devices.
- Recitation 1x week (Fr).
  - Lab-like, very helpful for homework.
  - Bring your laptops.
- Homework, midterm exams, project.

Communication

- We have a website and a Canvas site, with Piazza enabled.
  - Follow the link from the main Canvas page/syllabus to sign up for Piazza.
- Please:
  - Use Piazza to communicate with us as much as possible, unless the matter is sensitive.
  - Make your questions *public* as much as possible, since that's the literal point of Piazza.
- We have office hours! Or, by appointment.

"How do I get an A?"

- 10% in-class participation and exercises
- 40% homework
  - Both written (proof-y) and coding (implementation-y).
  - First one (mostly coding) released!
- 30% two (2) midterm exams
  - Date of second one depends a bit on guest lecture scheduling; I will post it ASAP.
- 20% final project
  - There will be some options here.
- No final exam; exam slot used for project presentations.
- We have late days and a late day policy; read the syllabus.

CMU can be a pretty intense place.

- A 12-credit course is expected to take ~12 hours a week.
- I aim to provide a rigorous but tractable course.
  - More frequent assignments rather than big monoliths.
  - Two exams reduces the pressure of just a single exam.
- Please keep me apprised of how much time the class is actually taking and whether it is interfacing badly with other courses.
  - I have no way of knowing if you have three midterms in one week.
  - Sometimes, we misjudge assignment difficulty.
- If it’s 2 am and you’re panicking…put my homework down, send me an email, and go to bed.

What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

Our first representation: Abstract Syntax

- A tree representation of source code based on the language grammar.
- Concrete syntax: The rules by which programs can be expressed as strings of characters.
  - Use finite automata and context-free grammars, automatic lexer/parser generators
- Abstract syntax: a subset of the parse tree of the program.
- (The intuition is fine for this course; take compilers if you want to learn how to parse for real.)

WHILE abstract syntax

- Categories:
  - $S \in$ **Stmt** statements
  - $a \in$ **Aexp** arithmetic expressions
  - $x, y \in$ **Var** variables
  - $n \in$ **Num** number literals
  - $P \in$ **BExp** boolean predicates
  - $l \in$ **labels** statement addresses (line numbers)
- Syntax:
  - $S ::= x := a \mid \texttt{skip} \mid S_1 ; S_2$
    $\mid \texttt{if}\ P\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ P\ \texttt{do}\ S$
  - $a ::= x \mid n \mid a_1\ op_a\ a_2$
  - $op_a ::= + \mid - \mid * \mid / \mid \ldots$
  - $P ::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ P \mid P_1\ op_b\ P_2 \mid a1\ op_r\ a2$
  - $op_b ::= \texttt{and} \mid \texttt{or} \mid \ldots$
  - $op_r ::= < \mid \leq \mid = \mid > \mid \geq \mid \ldots$

Concrete syntax is similar, but adds things like (parentheses) for disambiguation during parsing

Example WHILE program

```
y := x;
z := 1;
while y > 1 do
   z := z * y;
   y := y – 1
```

Exercise: Building an AST

```
y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y – 1
```

Exercise: Building an AST for C code

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

Our first static analysis: AST walking

- One way to find "bugs" is to walk the AST, looking for particular patterns.
  - Walk 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.
  - Tension between language agnosticism and semantic information available.
  - Consider "grep": very language agnostic, not very smart.
- One common 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 find a bug in AST element of type X, override visitX
- Other more recent approaches based on semantic search, declarative logic programming, or query languages.

Example: shifting by more than 31 bits.

```
For each instruction I in the program
  if I is a shift instruction
    if (type of I's left operand is int
        && I's right operand is a constant
        && value of constant < 0 or > 31)
      warn("Shifting by less than 0 or more
           than 31 is meaningless")
```

Dashboard / Java queries

Inefficient empty string test

https://help.semmle.com/wiki/display/JAVA/Inefficient+empty+string+test

Created by Documentation team, last modified on Mar 28, 2019

Name: Inefficient empty string test

Description: Checking a string for equality with an empty string is inefficient.

ID: java/inefficient-empty-string-test

Kind: problem

Severity: recommendation

Precision: high

Query: InefficientEmptyStringTest.ql › Expand source

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.

```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

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."
```

Practice: String concatenation in a loop

- Write pseudocode for a simple syntactic analysis that warns when string concatenation occurs in a loop
  - In Java and .NET it is more efficient to use a StringBuffer
  - Assume any appropriate AST elements

WHILE abstract syntax

- Categories:
  - $S \in$ **Stmt** statements
  - $a \in$ **Aexp** arithmetic expressions
  - $x, y \in$ **Var** variables
  - $n \in$ **Num** number literals
  - $P \in$ **BExp** boolean predicates
  - l $\in$ **labels** statement addresses (line numbers)
- Syntax:
  - S ::= x := a | skip | S_1 ; S_2
    | if P then S_1 else S_2 | while P do S
  - a ::= x | n | a_1 op_a a_2
  - op_a ::= + | - | * | / | …
  - P ::= true | false | not P | P_1 op_b P_2 | $a1$ op_r $a2$
  - op_b ::= and | or | …
  - op_r ::= < | ≤ | = | > | ≥ | ...

WHILE3ADDR: An Intermediate Representation

- Simpler, more uniform than WHILE syntax
- Categories:
  - I ∈ **Instruction** instructions
  - x, y ∈ **Var** variables
  - n ∈ **Num** number literals
- Syntax:
  - I ::= x := n | x := y | x := y op z | goto n | if x op_r 0 goto n
  - op_a ::= + | - | * | / | …
  - op_r ::= < | ≤ | = | > | ≥ | ...
  - P ∈ **Num** → I

Exercise: Translating to WHILE3ADDR

- Categories:
  - I ∈ **Instruction** instructions
  - x, y ∈ **Var** variables
  - n ∈ **Num** number literals

- Syntax:
  - I ::= x := n | x := y | x := y op z
    | goto n | if x op_r 0 goto n
  - op_a ::= + | - | * | / | …
  - op_r ::= < | ≤ | = | > | ≥ | ...
  - P ∈ **Num** → I

While3Addr Extensions (more later)

- Syntax:
  - I ::= x := n | x := y | x := y *op* z
    | goto n | if x op_r 0 goto n
    | x := $f(y)$
    | return x
    | x := $y.m(z)$
    | x := $\&p$
    | x := $*p$
    | $*p$:= x
    | x := $y.f$
    | $x.f$:= y

For next time

- Get on Piazza and Canvas
  - Answer our quizzes about office hours!
- Read lecture notes and the course syllabus
- Homework 1 is released, and due next Thursday.