Program Analysis

CS350 Introduction to Software Engineering

Shin Yoo

Analysis *noun.* a detailed examination of the elements or structure of something.

Analysing a Program… for what purpose?

- Many different objectives can be considered, for example (all real):
  - Violation of any Intellectual Property and Licensing
  - Authorship for software forensic
- But two major objectives are:
  - Correctness: you want to analyze and check specific properties to argue and reason about program correctness
  - Optimisation: you want to extract information required for optimization, or to ensure that optimization does not affect correctness

Static vs. Dynamic

- **Static Analysis:** your analysis is simply based on source code, without any actual execution
  - Examples: compiler optimisation, static code warning, linters
- **Dynamic Analysis:** your analysis includes information extracted from concrete executions of the target program
  - Examples: software testing

Static vs. Dynamic

- **Static Analysis:** considers program behavior against all possible inputs.
  - Consequently, this is **over-approximation** of actual program behavior (because certain input may not ever be used in concrete executions).
  - Accuracy is limited due to scope of analysis and differences that can only be captured dynamically.
- **Dynamic Analysis:** considers only the actual executions observed.
  - Consequently, this is **under-approximation** of the actual program behavior (because inputs we haven’t seen may result in different outcome).
  - Accuracy is limited because of executions you did not observe.

Dynamic Under-Approximation

- Potential division by zero error that we can plainly see in the source code
- Without an input that triggers the condition, we cannot find out dynamically

```
def compute_density(a, b):
    return a / b

def test_compute_density():
    assert compute_density(1, 2) == 0.5
    assert compute_density(2, 8) == 0.25
```

Static Over-Approximation

- Intra-procedural analysis of `compute_density` will raise an alarm for division by zero
- However, if you consider the entire program, this may never happen

```
def compute_density(a, b):
    return a / b

def process_data(x, y):
    if y <= 0:
        do_something()
    else:
        d = compute_density(x, y)
        do_something_else(d)
```

Focus Today

- We will mainly discuss static analysis today.
- Program analysis is a concept that encapsulates both static and dynamic analysis, but the absolute majority of dynamic analysis is now software testing, which can be treated independently in itself.

Scopes of Static Analysis

- Intra-procedural Analysis: considers a single procedure (function, method) one by one in isolation
    - Simple, but often imprecise (e.g., our example)
- Inter-procedural Analysis: analysis is performed over the entire program
    - Inlining can be used to make analysis inter-procedural
    - A key issue is alias analysis: the same memory address can be referred to by different variable names at different locations, affecting the precision

Context Sensitivity

- Context-insensitive Analysis: ignores the calling context. In the example, d is either 3, 8, 13, or 18. That is, it knows there are three calls from `bar` to `foo` with arguments 0, 5, and 5, but cannot differentiate them.
- Context-sensitive Analysis: can differentiate the three calls. The value of d is 18.

```
def foo(x):
    return x + 1

def bar():
    a = foo(0)
    b = foo(5)
    c = foo(5)

    d = a + b + c # what is d here?
```

Other forms of polyvariance

- Polyvariance: the degree to which an analysis structurally differentiates approximations of program values
    - Flow-sensitivity: can account for the order of statements, e.g., “after line X, variable `a` and `b` point to the same memory address”
    - Path-sensitivity: can account for branch predicates and the resulting paths, e.g., “in the false branch of `if x > 0:`, `x` is assumed to be less then or equal to 0”

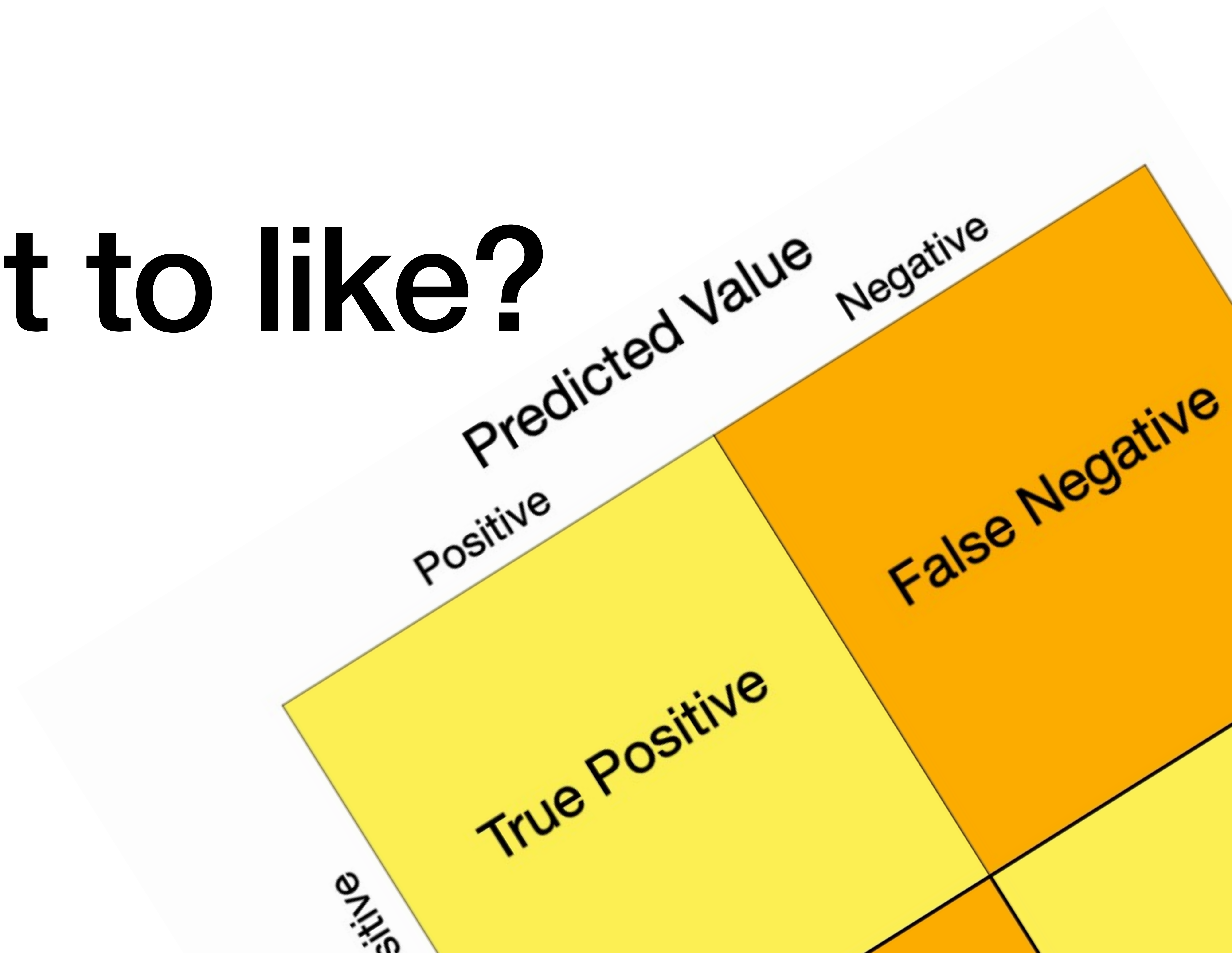
Types of Static Program Analysis

- Abstract Interpretation: maps program states to specific properties via abstraction
- Symbolic Execution: executes the program symbolically, i.e., with groups of inputs that share the same execution path
- Formal Verification: converts both the program and the property you want to prove into sets of logical statements, and proves the satisfiability of the conjunction of both (i.e., $F_{program} \land \neg F_{error}$)

Types of Static Program Analysis

- Pointer Analysis / Shape Analysis: analyses which pointer/heap reference can point to which actual memory locations.
- Dataflow Analysis: analyses information about sets of values calculated at various points in program, e.g., liveliness analysis, forward/backward dependence analysis, constant propagation…
- Program Slicing: a specific instance of dependence analysis that identifies parts of program that are related to a specific variable at a specific location
- Dead Code Elimination: identifies and removes code that cannot affect program execution

Okay, what is not to like?



False Positives

- “Positive” here means that the source code actually has an issue
- “False Positive”: static analysis predicts an issue, but the prediction is based on over-approximation, therefore the prediction is “false” —> false positive
- Why is this bad?

Actual Value \ Predicted Value	Positive	Negative
Positive	True Positive	False Negative
Negative	False Positive	True Negative

Let’s look at a few real world examples…

Classifying False Positive Static Checker Alarms in Continuous Integration Using Convolutional Neural Networks, Lee, S., Hong, S., Yi, J., Kim, T., Kim, C. and Yoo, S., *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 391–401.

HANDLE_LEAK

- A: `ret` is zero only when `dynamic_load` fails. However, the static analysis does not know this semantic, and raises `HANDLE_LEAK` warning at line 8.
- B: `_close(fd)` in line 18 does release the handle, but the static analysis does know know this domain-specific function and its semantic.

(a) A false alarm with a value-sensitive error-handling path

```
01: int create_file_attr() {
...
05:   ret = dynamic_load(&func_handle);
    //acquisition
06:   if (ret == 0) {
07:     debug("loading error\n");
08:     return FILE_ERROR;   // expiration
```

(b) A false alarm with a domain-specific resource-release operation

```
11: int write_profile() {
12:   fd = open(fpath, O_RDWR); // acquisition
...
18:   _close(fd);
19:   return n; }  // expiration
```

Fig. 2: Examples of false alarms from the Resource Handle Leak checkers

DEREF

Null-pointer Dereference

- Analysis thinks that after line 1-2, `obj_list` can be potentially `null` (because there is the check at line 1)
- So it raises null-pointer dereference at line 5
- However, `g_list_append` accepts `null` as a valid input that represents an empty list

```
01: if (obj_list)
02:   length = g_list_length(obj_list);
03: for (i = length; i < capacity; i++){
04:   obj = g_new(obj_t, 1);
05:   obj_list = g_list_append(obj_list, obj);
06: }
```

Fig. 4: A Null Pointer Dereference After Null false alarm

DOUBLE_FREE

- Static analysis raises an alarm if it thinks free is called twice on two aliases to the same memory location.
- In the example, the analysis misses the alias update that takes place at line 3.
- Consequently, it considers line 5 as a double free.

```
01: len = length(node_list)
02: for (i = 0; i < len; i++) {
03:   node = get_element_at(node_list, 0);
04:   node_list = node_list->next;
05:   node_free(node);
06: }
```

Fig. 3: A Double Free false alarm

Common Limitations

- In all previous examples, it seems that the weak analysis itself is the problem - scope/context of analysis, lack of knowledge about semantics of domain specific functions, imprecise alias analysis, etc…
- But are those all the reasons? We can make perfect static analysis tools if we just try harder?
- Let’s look at a more fundamental example.

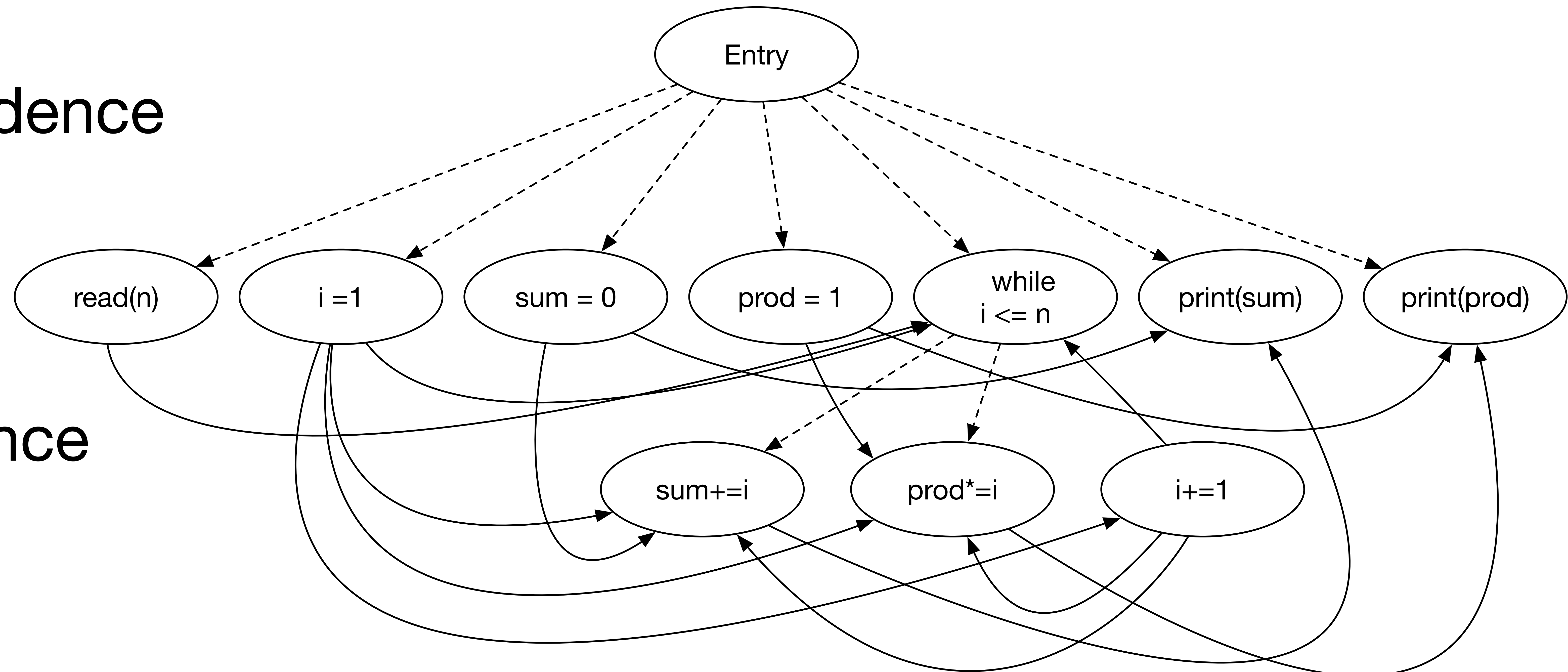
Program Slicing

- Produces a subset of the given program that is related to the computation of a value at a specific location
  - (value, location) is called the slicing criterion
  - Useful for debugging, program comprehension, reuse…

```
def func:
    read(n)
    i = 1
    sum = 0
    prod = 1
    while i <= n:
        sum += i
        prod *= i
        i += 1
    print sum
    print prod # slice here
```

Static Program Slicing

- Construct Program Dependence Graph (PDG)
  - Vertices: statements
  - Edges: control dependence (dashed lines), data dependence (solid lines)
- Slicing: start from the criterion, get the backward closure!



Now consider this program

- Predicate p and q only depends on `i` and `c`, respectively; `f()` and `g()` return constant values.
- `x@11` data-depends on `x@6`, which control-depends on `c@5`, which control depends on both `c@7` and `i@3`. `i@3` data-depends on `i@9`.
- Nothing to delete!
- Is it? :)

```
01:int mug(int i, int c, int x)
02:{
03:     while(p(i))
04:     {
05:         if(q(c)){
06:             x = f();
07:             c = g();
08:         }
09:         i = h(i);
10:     }
11:     printf("@%d\n", x); //slice here for x
12:}
```

Now consider this program

- If `q(c)` is initially false, it remains false (as we never enter the body of if). Also, `c` is not written over, so x retains its original value.

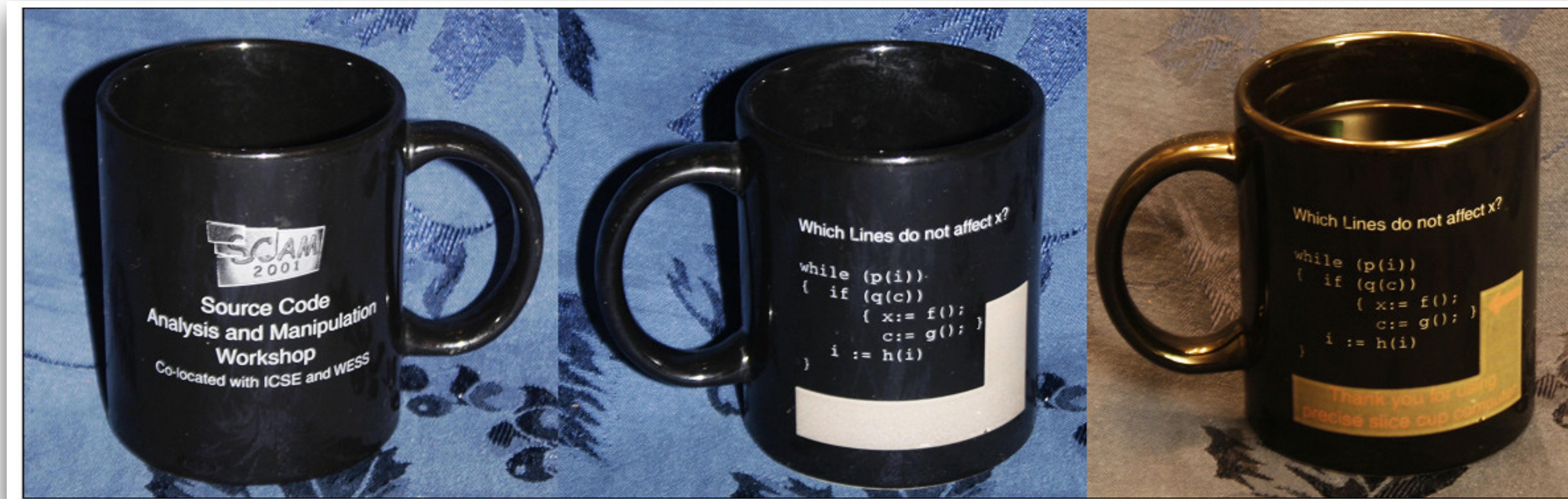
- If `q(c)` is true one or more times, x will be overwritten with `f()` in line 6. As long as this happens at least once, it is not important how often `q(c)` is true. Consequently, line 7 DOES NOT affect x in line 11!

```
01:int mug(int i, int c, int x)
02:{
03:    while(p(i))
04:    {
05:        if(q(c)){
06:            x = f();
07:            c = g();
08:        }
09:        i = h(i);
10:    }
11:    printf("@%d\n", x); //slice here for x
12:}
```

The (in)famous SCAM Mug

Source Code Analysis and Manipulation (SCAM) Workshop 2001

- Static slicing cannot get the minimum slice of this program.
- But the souvenir from SCAM Workshop in 2001 can slice it :)



Rice's Theorem

The Fundamental Limitation

- Any non-trivial semantic properties of programs are undecidable.
- Proof sketch by reduction to halting problem: suppose we have an algorithm that can decide whether a given program p has a property q. Then we can construct a solution to the halting problem…

```
def halt(a, i):
    def t():
        a(i)
        return
    return does_it_have_q(t)
```

```
def halt(a, i):
    def t(n):
        a(i)
        return n * n
    return does_it_compute_squares(t)
```

Does this mean that static analysis is useless? NO.

- Theoretical limitation against arbitrary programs and inputs does not prevent us these techniques from being useful to many practical programs.
- Many industrial-strength static analysis tools exist.

Clang Static Analyzer

Infer by Facebook

- A static analysis tool for Java, C++, Objective C, and C
- Written in OCaml
- https://github.com/facebook/infer

Static Bug Finder Tools (e.g., SpotBugs)

- These tools have a large catalogue of potential programming errors, and statically scan your program to match them
- SpotBugs, earlier known as FindBugs, is a well known such tool for Java: its bug description page can be found here: https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html

Lint

- Static analysis tools that flags up programming errors, bugs, stylistic/coding convention errors, and suspicious patterns.
- Typically more useful for dynamically typed languages, such as JavaScript and Python, as compilers/runtimes do much less checking

Popular Lints

- Python: PyLint (https://github.com/pylint-dev/pylint), flake8 (https://github.com/PyCQA/flake8)
- JavaScript: JSLint (https://www.jslint.com/), ESLint (https://eslint.org/)

Style Guides

- Tells a programmer how to work with a specific programming language
- Rules can range from simply formatting rules and naming convention to common bugs that can be checked statically
- PEP 8: Python Style Guide (https://peps.python.org/pep-0008/)
- MISRA C: development guideline for C maintained by MISRA Consortium, especially for embedded systems (https://www.misra.org.uk/)

Summary

- Program Analysis aims to automatically analyse program behaviour.
- Static program analysis is cheap (no execution) and typically used as a universal quality filter (e.g., lint in CI/CD pipeline).
- General static analysis tools/bug detectors can analyse the program for common errors (e.g., null pointer deference), but cannot check for any errors in the business logic (i.e., any semantic that is specific only to your program)
- If you're interested :)
  - CS524 Program Analysis (Prof. Kihong Heo)
  - CS492 Program Reasoning (Prof. Kihong Heo)