Workshop on Dynamic Analysis, Portland, Oregon, 2003

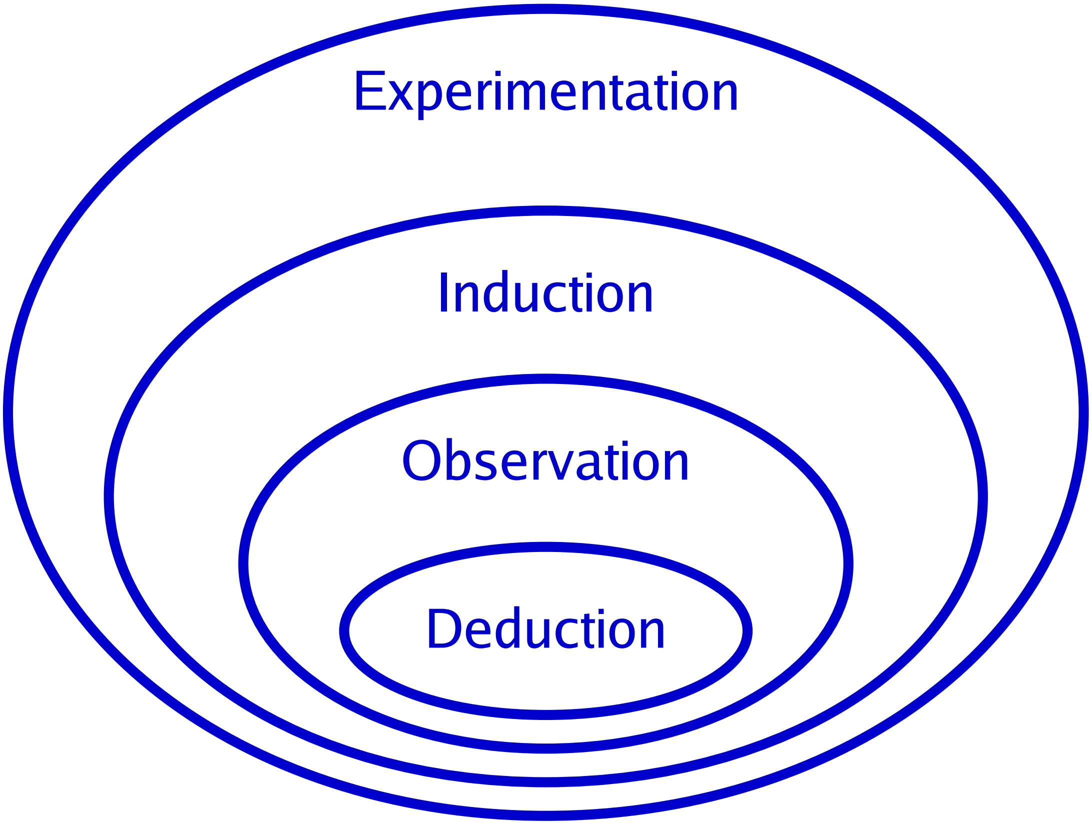
# *Program Analysis: A Hierarchy*

Andreas Zeller

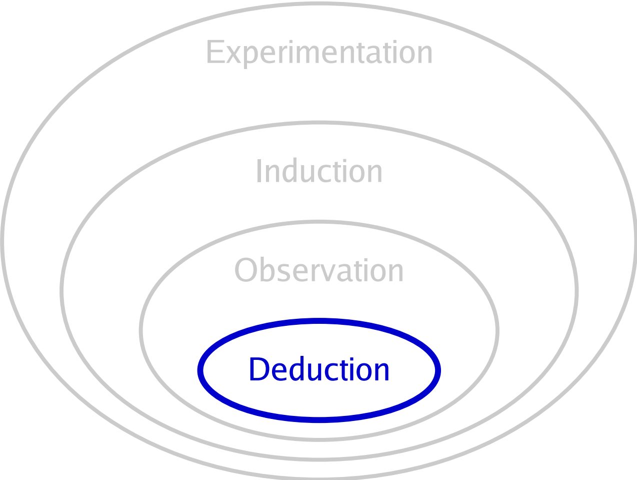
Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

# *A Hierarchy of Reasoning*





# *Deductive (static) Program Analysis*




**Deduction:** reasoning from from the *general* to the *particular*

- does not execute any programs (hence "static")
- abstracts from actual runs
- can thus determine properties that hold for *all* runs and *all* embeddings

Traditional domain: logic, *program optimization* in compilers

**Examples:** Control and data flow analysis · symbolic interpretation · *program slicing*

# *Example: Program Slicing*



```
char *format = "a = %d";
if (p)
    a = compute_value();
sprintf(buf, format, a);
```

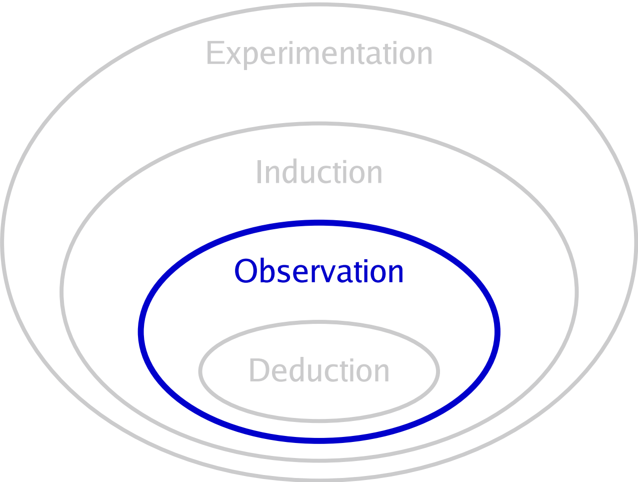
Assume we find `"a = 0"` in `buf`. What’s the cause?

In deductive analysis, two variables are *dependent* on each other if one can affect the other’s value:

- `buf` is data dependent on `format` and `a`
- `a` is control dependent on `p` . . .

Dependency is undecidable: *conservative approximation*

# *Observational Program Analysis*




**Observation:** finding *facts*

- observes a single run of the program (hence "dynamic")
- finds *irrefutable facts* about the observed run
- facts hold
  for observed run only
- can make use of deduction

Traditional domain: *metrics*

**Examples:** Debuggers · coverage tools · *dynamic slicing*

# *Example: Dynamic Slicing*

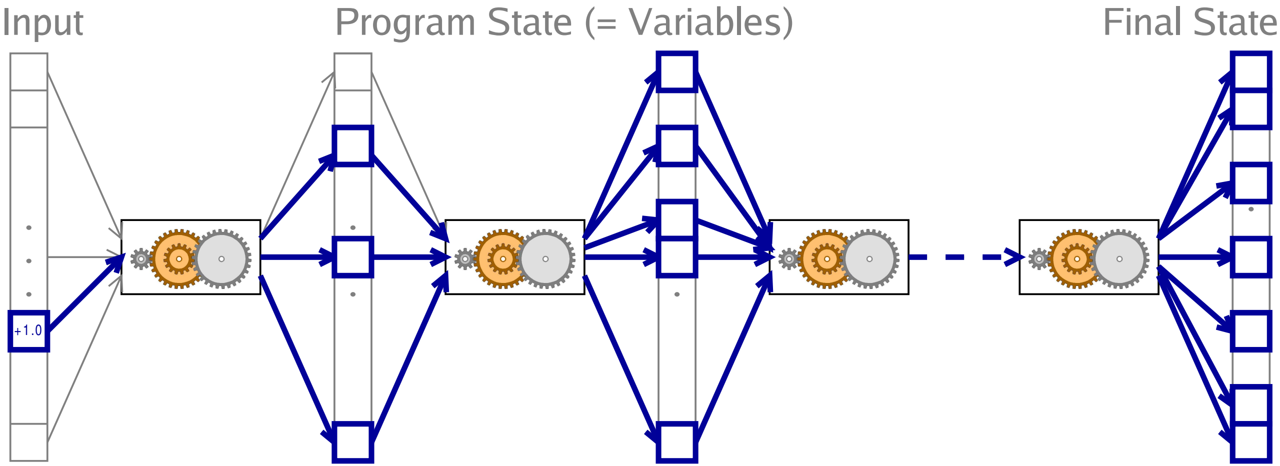
```
char *format = "a = %d";
if (p)
    a = compute_value();
sprintf(buf, format, a);
```

Still, we find `"a = 0"` in `buf`. What's the cause?

Assume we also observe that `p` is true. Then, dynamic slicing can deduce that `a`'s value stems from `compute_value()`.

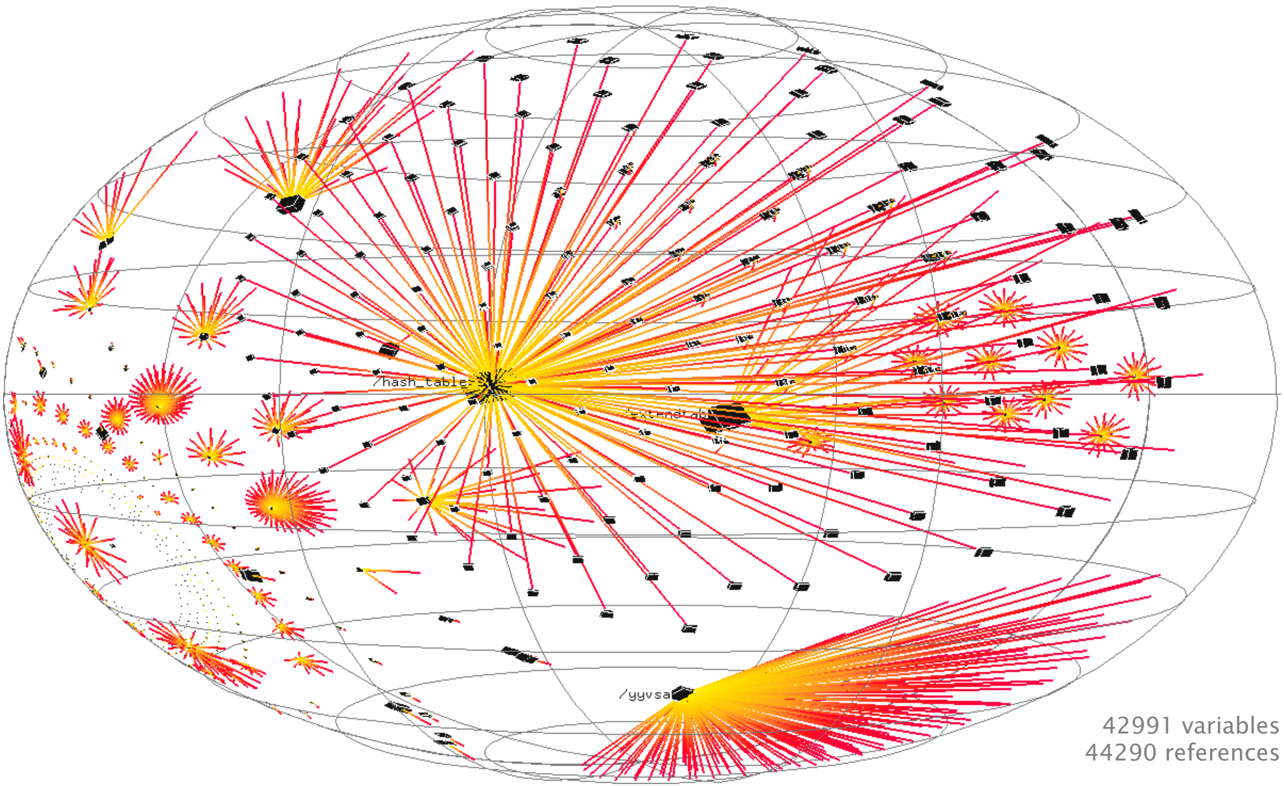
# *Observing Time*

The effects of variable values *accumulate* during execution – the longer the time span observed, the more effects




This “short-sightedness” affects *static* and *dynamic* slicing.

# *Observing Space*

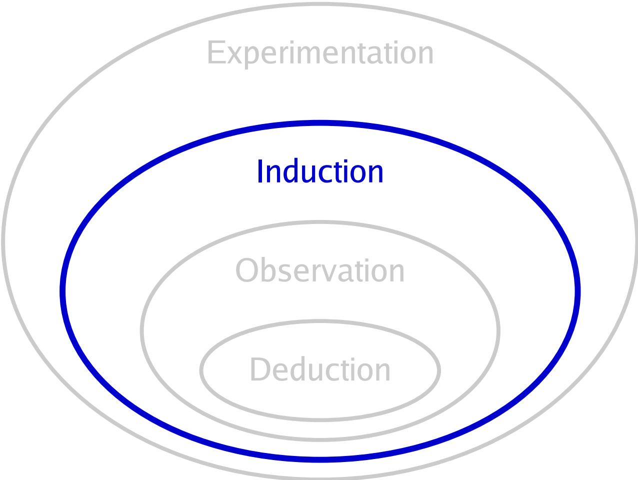



42991 variables
44290 references

897 variables (≤ 2%) are affected by a change

# *Inductive Program Analysis*






**Induction:** reasoning from the particular into the abstraction

- observes *multiple runs*
- finds *commonalities* and *anomalies* across runs
- findings hold for observed runs only
- must use observation; can use deduction

Traditional domain: *natural science*

**Examples:** Coverage comparison · relative debugging · *dynamic invariant detection*

# *Example: Invariant Detection*

```
3 char *format = "a = %d";
4 if (p)
5     a = compute_value();
6 sprintf(buf, format, a);
```

We execute the code under several random inputs and flag an error each time `buf` contains `"a = 0"`.

An invariant detector can then determine that, say,

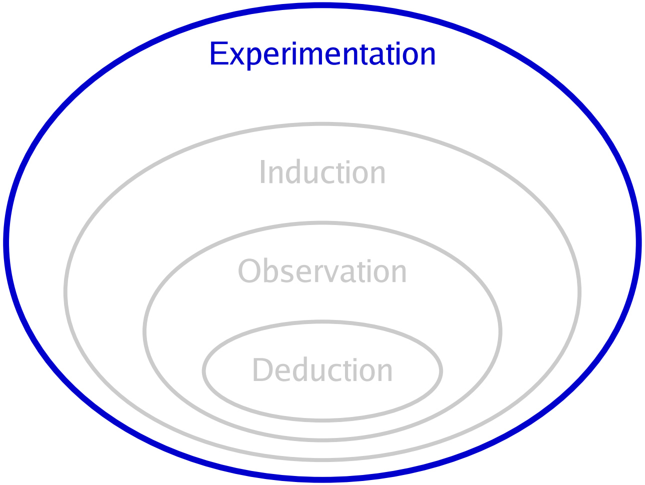
```
a < 2054567 || a % 2 == 1
```

holds at line 6 for all runs where the error occurs.

Obviously, something very strange is going on.

# *Experimental Program Analysis*






**Experimentation:** conducting experiments based on prior findings

- executes and *controls* multiple runs
- narrows down *causes*
- must use observation; can use deduction and induction

Traditional domain: *experimental science*

**Examples:** Delta debugging · *Experiments by humans*

# *Example: Experiments*

```
char *format = "a = %d";
if (p)
     a = compute_value();
sprintf(buf, format, a);
```

The failure occurs for most values of `a`:
`a` cannot be the cause for `buf` being `"a = 0"`.

The only remaining cause is `format`, and indeed:

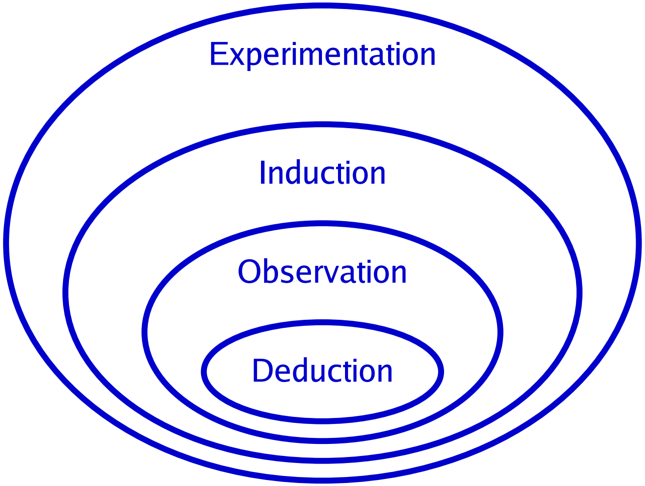
```
double a;
```

Altering `format` to `"a = %f"` fixes the failure
(and *proves* that `format` was the failure cause)

Delta debugging can isolate such causes automatically by *narrowing the difference* between a failing and non-failing run.

# *Conclusion and Consequences*





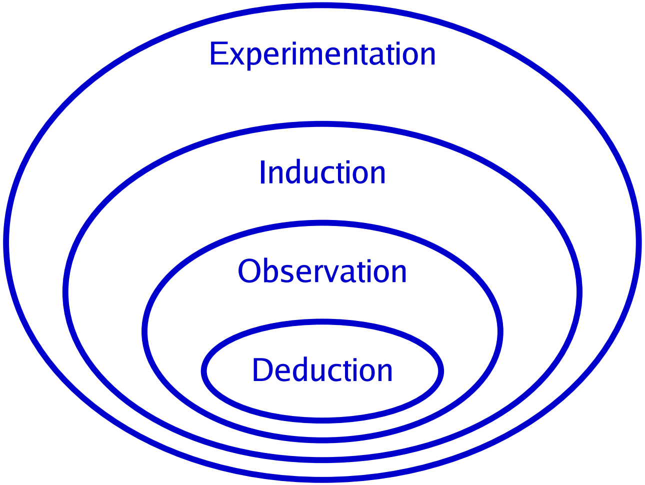

Each class of program analysis

- is *defined* by the # of runs considered (from 0 to ∞)
- can *use* "inner" classes (but not vice versa)
- is *limited* in its findings by the underlying reasoning technique:

- To determine *causes*, one needs experiments.
- To *summarize* findings, one must induce over $n$ runs.
- To find *facts*, one needs observation.
- Deduction (surprise?) cannot tell any of these!

# *Topics to Talk About*






- How can we better *leverage* the findings of "inner" classes for "outer" classes?
- What other *induction* methods (data mining, machine learning, ...) could be used?
- How can we leverage *experimentation* (e.g. generate runs that satisfy given properties)?

- What are the *practical limits* of the individual classes?
- What are the typical *uses* of dynamic analysis?
- Does this hierarchy make sense?