Dynamic Program Analysis

Nataliia Stulova

SMA: Software Modeling and Analysis

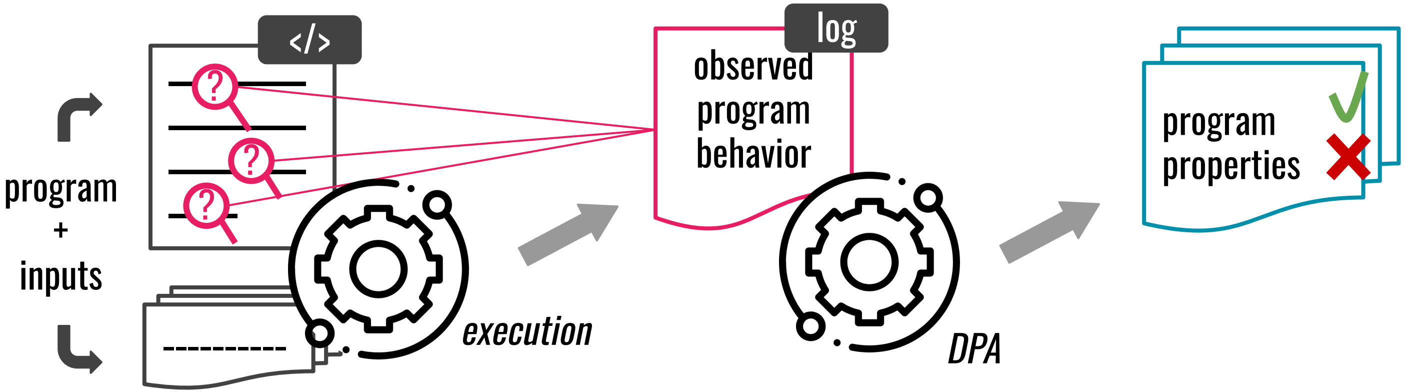
b
UNIVERSITÄT
BERN

Roadmap

- **what is dynamic analysis?**
- program instrumentation
- dynamic analysis use cases:
  - understanding program performance
  - contracts and program correctness beyond types

dynamic program analysis

Dynamic program analysis is the **analysis of program properties** by observing the **program behavior** during execution (on a concrete architecture) with concrete inputs.



dynamic VS static analyses

	Dynamic analysis	Static analysis
Information	execution behavior	program structure
Scope	executed program part	whole program
Soundnes	feasible (only FN)	feasible (either FP or FN)
Completeness	difficult	feasible
Imprecision source	limited inputs	abstractions
Scalability	easy	hard

FP = false positives, FN = false negatives

dynamic VS static analyses

	Dynamic analysis	Static analysis
Information	execution behavior	program structure
Scope	executed program part	whole program
Soundnes	feasible (only FN)	feasible (either FP or FN)
Completeness	difficult	feasible
Imprecision source	limited inputs	abstractions
Scalability	easy	hard

FP = false positives, FN = false negatives

dynamic VS static analyses

	Dynamic analysis	Static analysis
Information	execution behavior	program structure
Scope	executed program part	whole program
Soundnes	feasible (only FN)	feasible (either FP or FN)
Completeness	difficult	feasible
Imprecision source	limited inputs	abstractions
Scalability	easy	hard

FP = false positives, FN = false negatives

dynamic VS static analyses

	Dynamic analysis	Static analysis
Information	execution behavior	program structure
Scope	executed program part	whole program
Soundnes	feasible (only FN)	feasible (either FP or FN)
Completeness	difficult	feasible
Imprecision source	limited inputs	abstractions
Scalability	easy	hard

FP = false positives, FN = false negatives

dynamic VS static analyses

	Dynamic analysis	Static analysis
Information	execution behavior	program structure
Scope	executed program part	whole program
Soundnes	feasible (only FN)	feasible (either FP or FN)
Completeness	difficult	feasible
Imprecision source	limited inputs	abstractions
Scalability	easy	hard

FP = false positives, FN = false negatives

dynamic VS static analyses

	Dynamic analysis	Static analysis
Information	execution behavior	program structure
Scope	executed program part	whole program
Soundnes	feasible (only FN)	feasible (either FP or FN)
Completeness	difficult	feasible
Imprecision source	limited inputs	abstractions
Scalability	easy	hard

FP = false positives, FN = false negatives

why analyzing programs at run-time? (1/2)

Case 1: you are not satisfied with the *precision* of static analysis (over-/under-approximations):

```
int f(int age) {
  if (age < 16) {
    …
  } else {
    return g(age);}
}
```

age: read from run time input

static analysis: looks at types, says age is an integer (-32,768 to 32,767)

realistically: age > 120 or age < 0 makes no sense

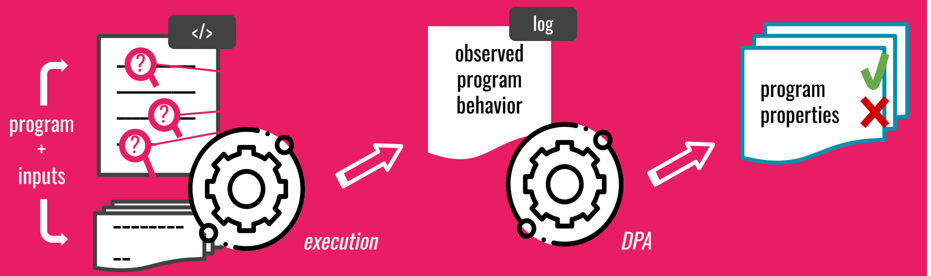
dynamic analysis: can observe exact values

why analyzing programs at run-time? (2/2)

Case 2: you are interested in detecting properties that are beyond the capabilities of static analysis:

- program “hot spots” - which parts of program take most resources?
- memory reference errors - is there uninitialized memory, indexing beyond array bounds, any leaks?
- likely invariants - what (implicit) properties actually hold for program variables and methods?

Roadmap



> what is dynamic analysis?

> **program instrumentation**

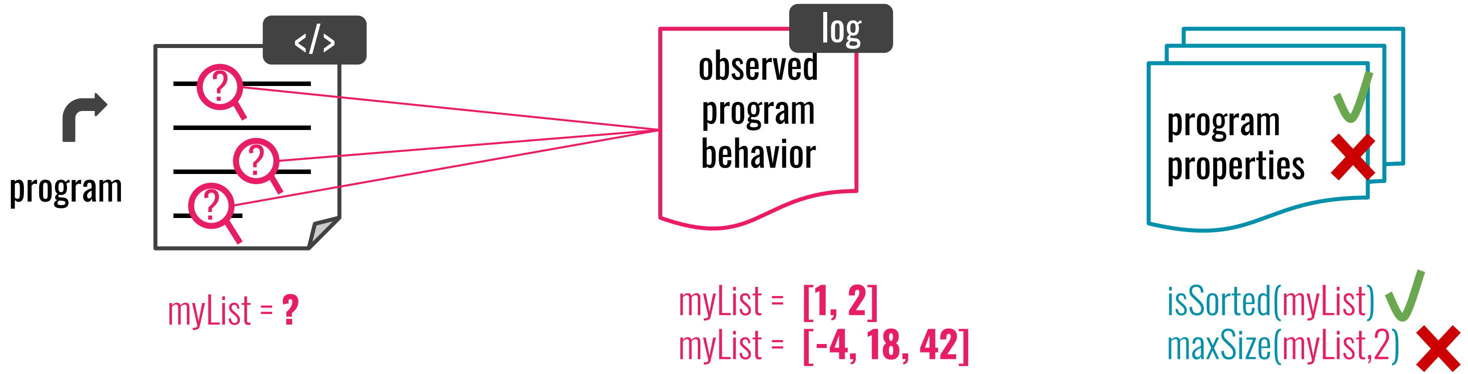
> dynamic analysis use cases:

>> understanding program performance

>> contracts and program correctness beyond types

program instrumentation

Instrumentation is a harness (special code) to capture run-time values of variables at points of interest:



DPA is inferring/checking program properties that hold at those points.

instrumentation points of interest

C++

```
void print_number(int* myInt) {
(1)  assert (myInt != NULL);
  printf ("%d\n",*myInt);
}

int main () {
  int a=10;
  int * b = NULL;
  int * c = NULL;
  b=&a;
(2)  assert (*b > 0);

  print_number (b);
  print_number (c);

(3)  assert (c != NULL)
  return *c;
}
```

- method entry (1)
  - captures values of input parameters
- program point (2)
  - captures values of specific variables
- method exit (3)
  - captures return values

what to consider for instrumenting

---

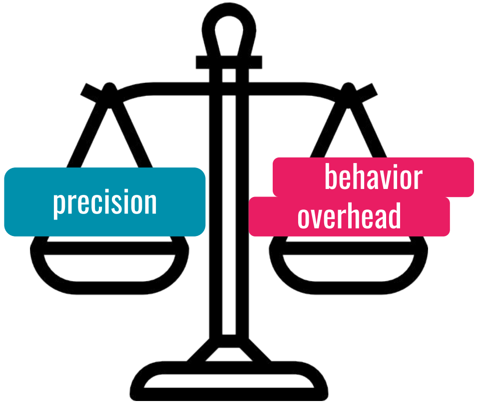
how much information is collected

which level is the instrumentation inserted at:

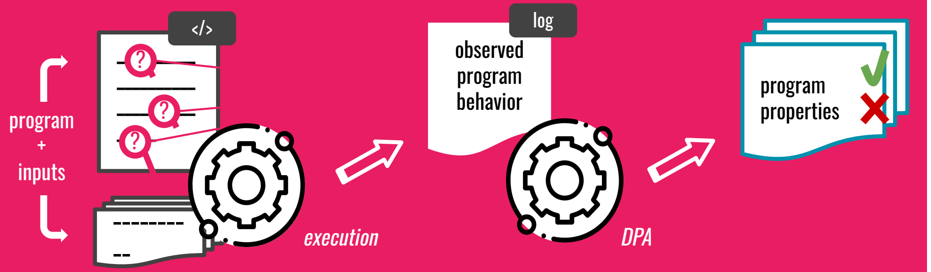
- from annotations in source code
- directly to object/byte/machine code

how intrusive the instrumentation is:

- performance overhead
- program behavior affected (esp. instrumentation that checks properties)



Roadmap



> what is dynamic analysis?

> program instrumentation

> dynamic analysis use cases:

>> **understanding program performance**

>> contracts and program correctness beyond types

performance profiling

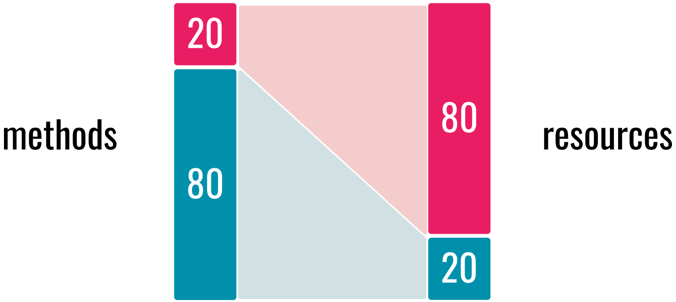
A form of dynamic program analysis that collects performance metrics of a program, usually done with a tool - a profiler.

Profilers:

- **event-based** - collect information at specified locations (high precision and overhead)
- **statistical** - collect information from run-time environment (less precise, but almost no overhead)

why studying program performance?

Pareto principle applies to programs too:



80% of the processor's time will be consumed by only 20% of the functions.

So assuming we have 100 functions, by just optimizing 20 of those, we can improve performance more than by optimizing all of the other 80 functions.

Additionally, high resource use may indicate bugs in the code

example: performance analysis guiding optimization

libraries

```
class A() {
  public int f(int x) {return x + 1;}
}

class B() {
  public int f(int x) {return x * 4;}
}
```

client

```
//...
Object ab;
if (g(0) == 1) ab = new (1));
else        ab = new B();

int c = g(1); (2)
int z = 0;

for(int i = c, i > 0, i--){  (3)
  z = z + ab.f(c);  (4)
}
```

Situation: source code of **g()** is not available from its library file.

What can we do with dynamic analysis?

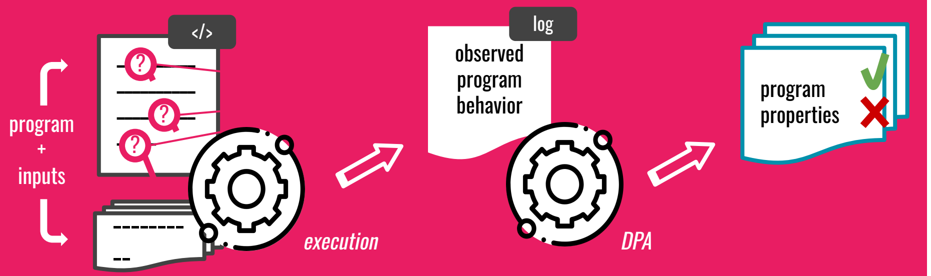
- observe values of **g(0)**, **g(1)** (1)(2)
- if they do not change, optimize:
  - if we can inline **f()** at (4), then we can speed up the loop at (3)

example: profiling memory usage in Python

Memory Profiler is a python module for monitoring memory consumption of a process as well as line-by-line analysis of memory consumption for python programs:

```
Line #    Mem usage    Increment  Occurrences   Line Contents
============================================================
     3   38.816 MiB   38.816 MiB           1   @profile
     4                                         def my_func():
     5   46.492 MiB    7.676 MiB           1       a = [1] * (10 ** 6)
     6  199.117 MiB  152.625 MiB           1       b = [2] * (2 * 10 ** 7)
     7   46.629 MiB -152.488 MiB           1       del b
     8   46.629 MiB    0.000 MiB           1       return a
```

Roadmap



> what is dynamic analysis?

> program instrumentation

> dynamic analysis use cases:

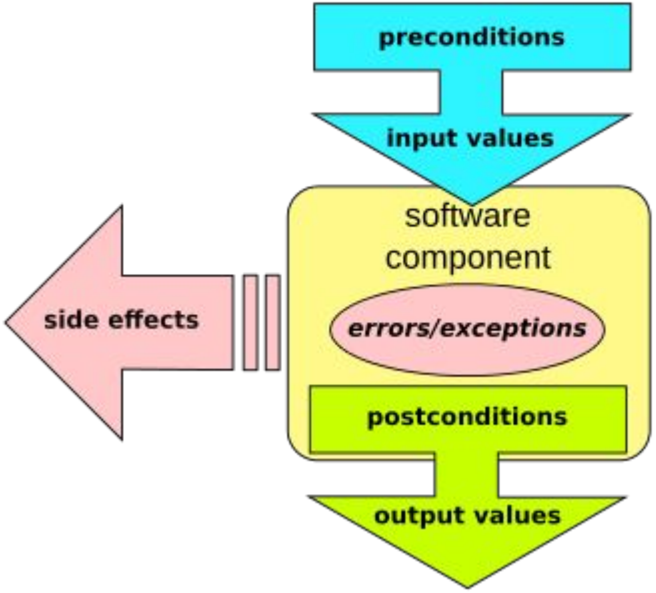
>> understanding program performance

>> **contracts and program correctness beyond types**

design by contract - to the types and beyond

The idea:

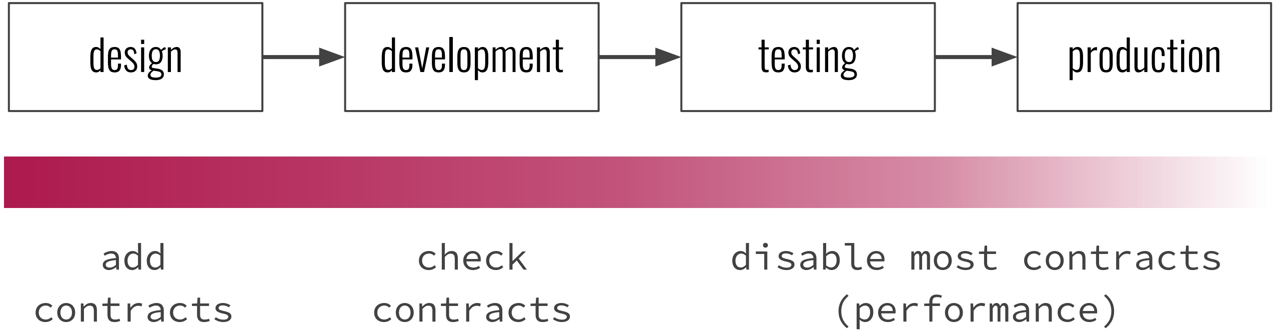
"...*software designers should define formal, precise and* ***verifiable*** *interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants.*



contracts place in software development process

---

a typical
program
lifecycle
+ contracts



native contract support

For many programming languages contract syntax is a part of the language and is understood by the compiler: Clojure, Kotlin, Scala, Spec#, …

Eiffel

```
put (x: ELEMENT; key: STRING) is
    -- Insert x so that it will be
    -- retrievable through key.
    require
       count <= capacity        } precondition
       not key.empty
    do
       ... Some insertion algorithm ...
    ensure
       has (x)
       item (key) = x            } postcondition
       count = old count + 1
    end
```

In this example:

- precondition: before inserting an element to a collection make sure there is space and the element key is non-null
- postcondition: after element insertion collection should have the element, specifically at a given key (no key collisions), and collection size grows by 1

third-party contract support

For some other programming languages contracts are enabled by a specialized tool, a pre-processor: C/C++/C#, Go, Java, Perl, PHP, Ruby, Rust, …

Java

```java
@Contract("null -> fail; _ -> param1")
```

@Contract, IntelliJ IDEA

method throws an exception if the first argument is null, otherwise it returns the first argument

Java

```java
/**
 * @pre f >= 0.0
 * @post Math.abs((return * return) - f) < 0.001
 */
public float sqrt(float f) { ... }
```

iContract

method calculates the square root of **f** within a specific margin of error (+/- 0.001).

instrumenting the contracts

- **preconditions**: state properties which should hold at method entry ①
- **postconditions**: state properties which should hold at method exit ③ (*optionally: for a given entry* ①)
- **invariants**: state properties which should hold at any point ①②③

C++

```
void print_number(int* myInt) {
①  assert (myInt != NULL);
   printf ("%d\n",*myInt);
}

int main () {
   int a=10;
   int * b = NULL;
   int * c = NULL;
   b=&a;
②  assert (*b > 0);

   print_number (b);
   print_number (c);

③  assert (c != NULL)
   return *c;
}
```

preconditions

Preconditions involve the system state and the arguments passed into the method before a method can execute.

java

```java
/**
 *  @pre f >= 0.0
 */
public float sqrt(float f)
{ ... }
```

- the precondition ensures that the argument f of function sqrt() is greater than or equal to zero.

postconditions

Postconditions involve the old system state, the new system state, the method arguments, and the method's return value.

java

```java
/**
 *  Append an element to a collection.
 *
 *  @post c.size() = c@pre.size() + 1
 *  @post c.contains(o)
 */
public void append(Collection c, Object o)
{ ... }
```

- the first postcondition specifies that the size of the collection must grow by 1 when we append an element. The expression c@pre refers to the collection **c** before execution of the append method.
- the second postcondition specifies that at the method exit **o** is a part of **c**

invariants

Invariants describe properties that hold at any given time during execution, so depending on their scope granularity they can be checked at program points, method boundaries, and class level:

java

```java
/**
 * A PositiveInteger is an Integer
 * that is guaranteed to be positive.
 *
 * @inv intValue() > 0
 */
class PositiveInteger extends Integer
{ ... }
```

- this class invariant guarantees that the PositiveInteger's value is always greater than or equal to zero. That assertion is checked before and after execution of any method of that class.

contracts and first order logic

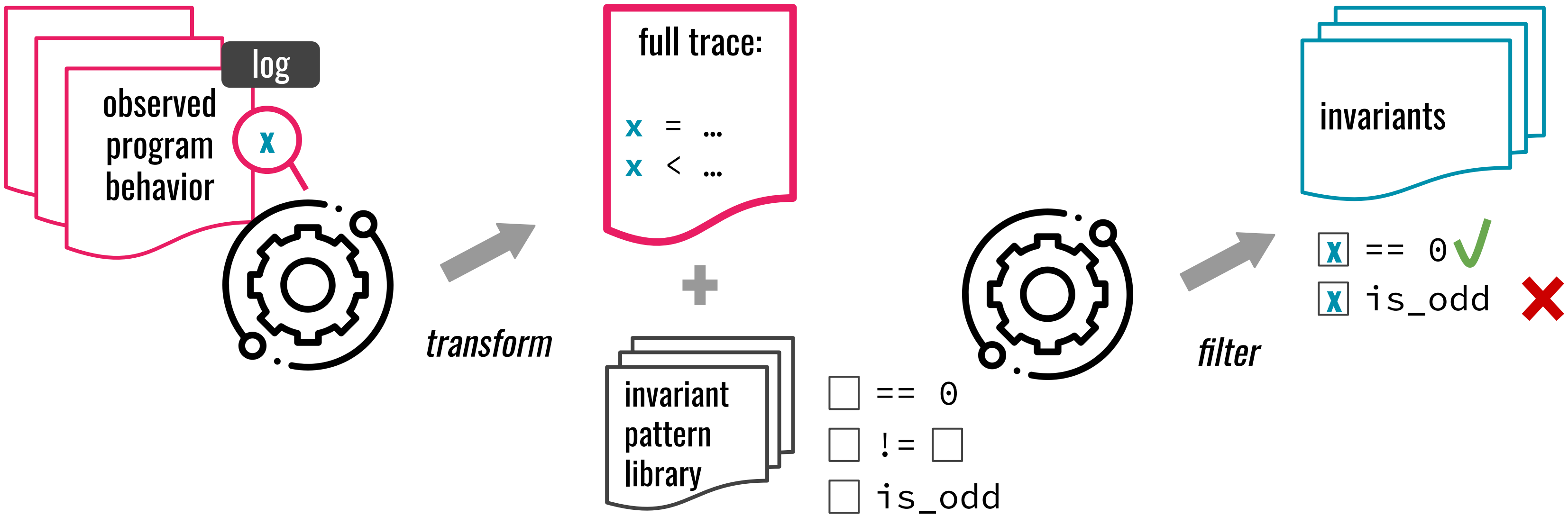
- quantifiers: forall, exists
- negation
- implication

java

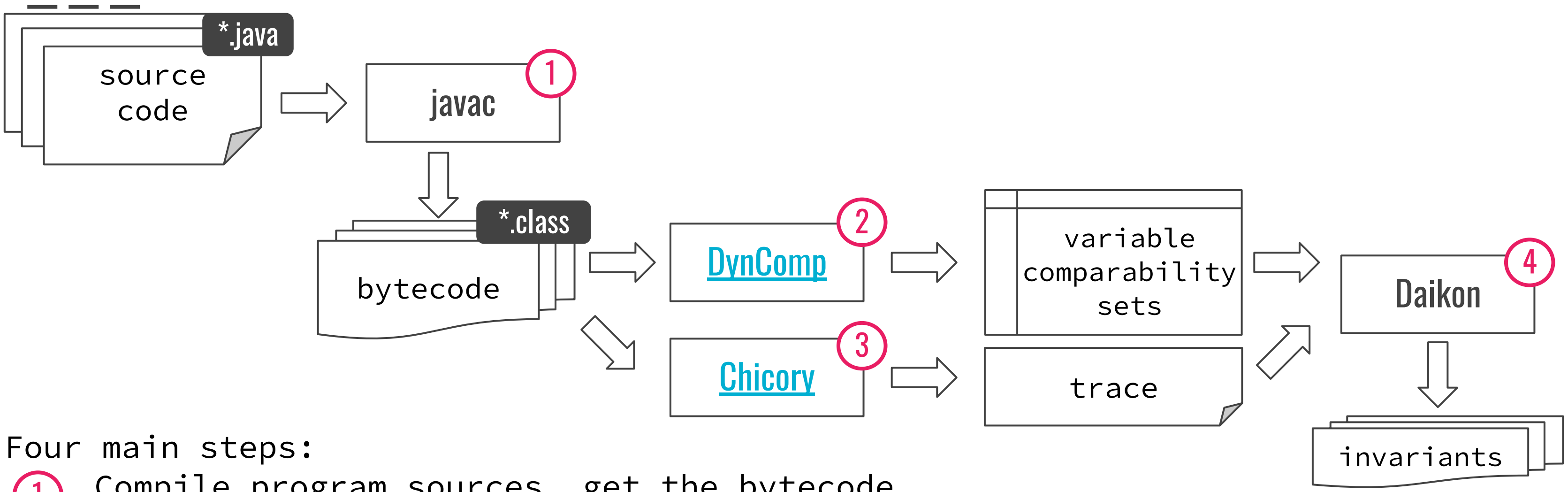
```java
/**
 *
 * A single office per employee.
 *
 * @invariant forall IEmployee e1 in getEmployees() |
 *                forall IEmployee e2 in getEmployees() |
 *                 (e1 != e2) implies e1.getOffice() != e2.getOffice()
 */
```

invariant detection with Daikon

Daikon is a tool for dynamic detection of likely invariants by M. Ernst et al.



Daikon for Java



Four main steps:

1. Compile program sources, get the bytecode
2. Run the program under DynComp component to group variables at each program point into comparability sets, limiting invariant scopes
3. Run the program under Chicory component to instruments the bytecode and produce the trace(s)
4. Analyze the trace(s) with Daikon to get invariants

demo!