Dynamic Contract Inference

Nadia Polikarpova

Software Verification
17.10.2012

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dynamic contract inference

- Location invariant – a property that always holds at a given point in the program

...
$x := 0$
...

$x = 0$

- Dynamic invariant inference – detecting location invariants from values observed during *execution*

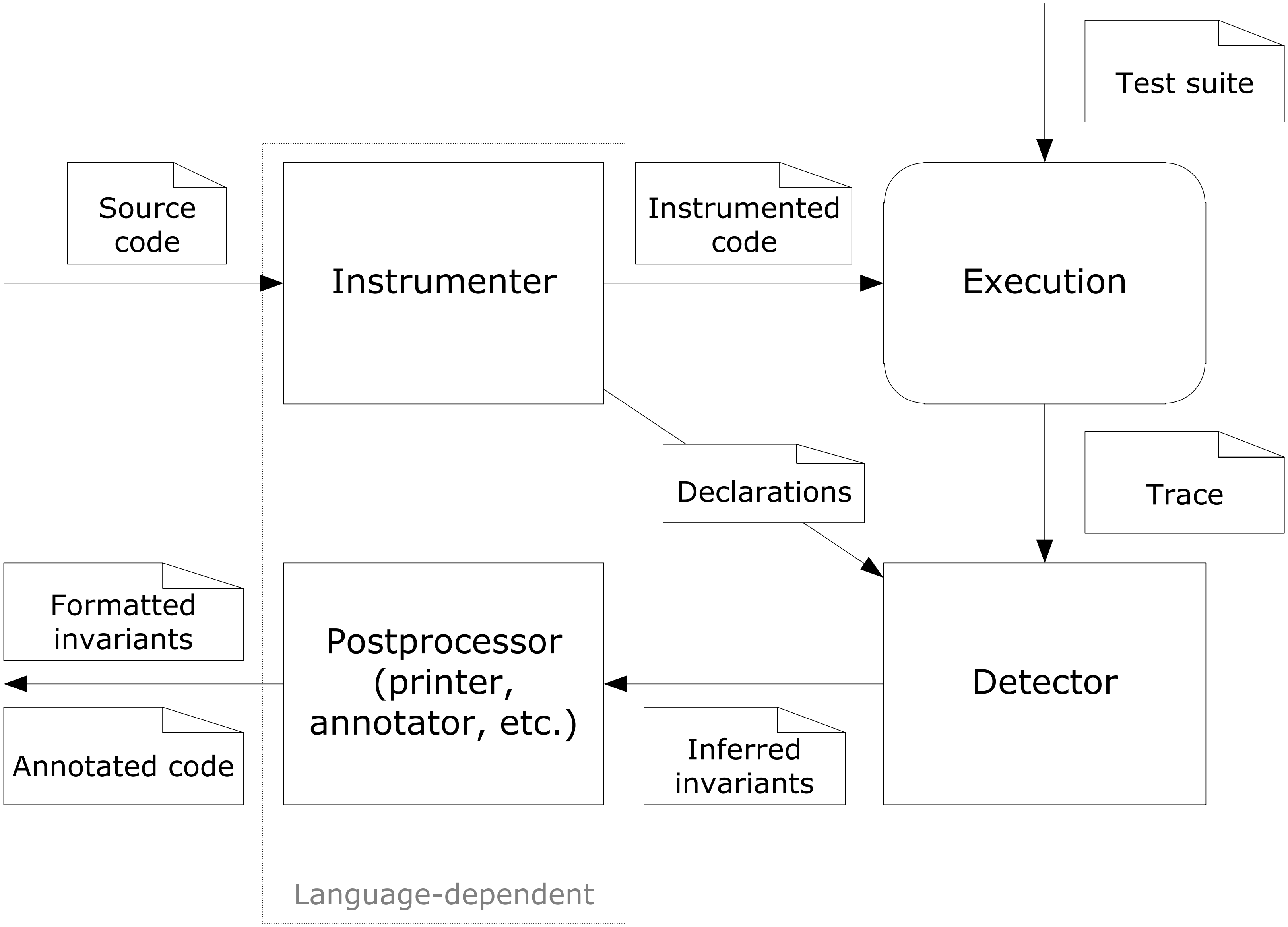
- Also called: invariant generation, contract inference, specification inference, assertion inference, ...

- Pioneered by Daikon
  http://groups.csail.mit.edu/pag/daikon/

Overview

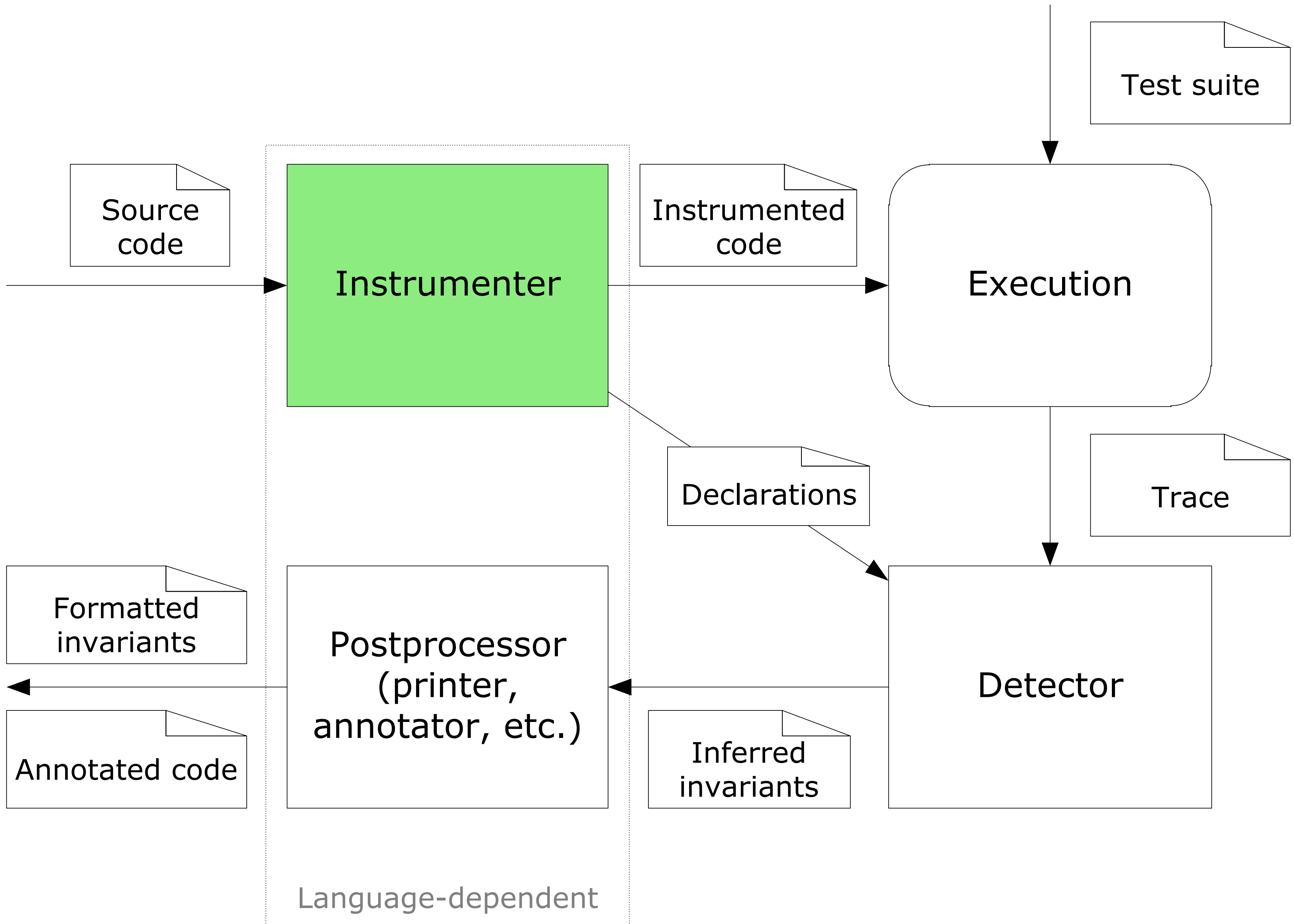
- How does Daikon work?
- Inferred invariants
- Improving inferred invariants
- Contract inference in Eiffel: CITADEL and AutoInfer

Daikon architecture



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Daikon architecture



Instrumenter

- Finds program points of interest
  - routine enter/exit, loop condition
- Finds variables of interest at these program points
  - current object, formals, locals, return value, expressions composed of other variables
- Modifies the source code so that every time a program point is executed, variable values are printed to the trace file

Instrumenter: example

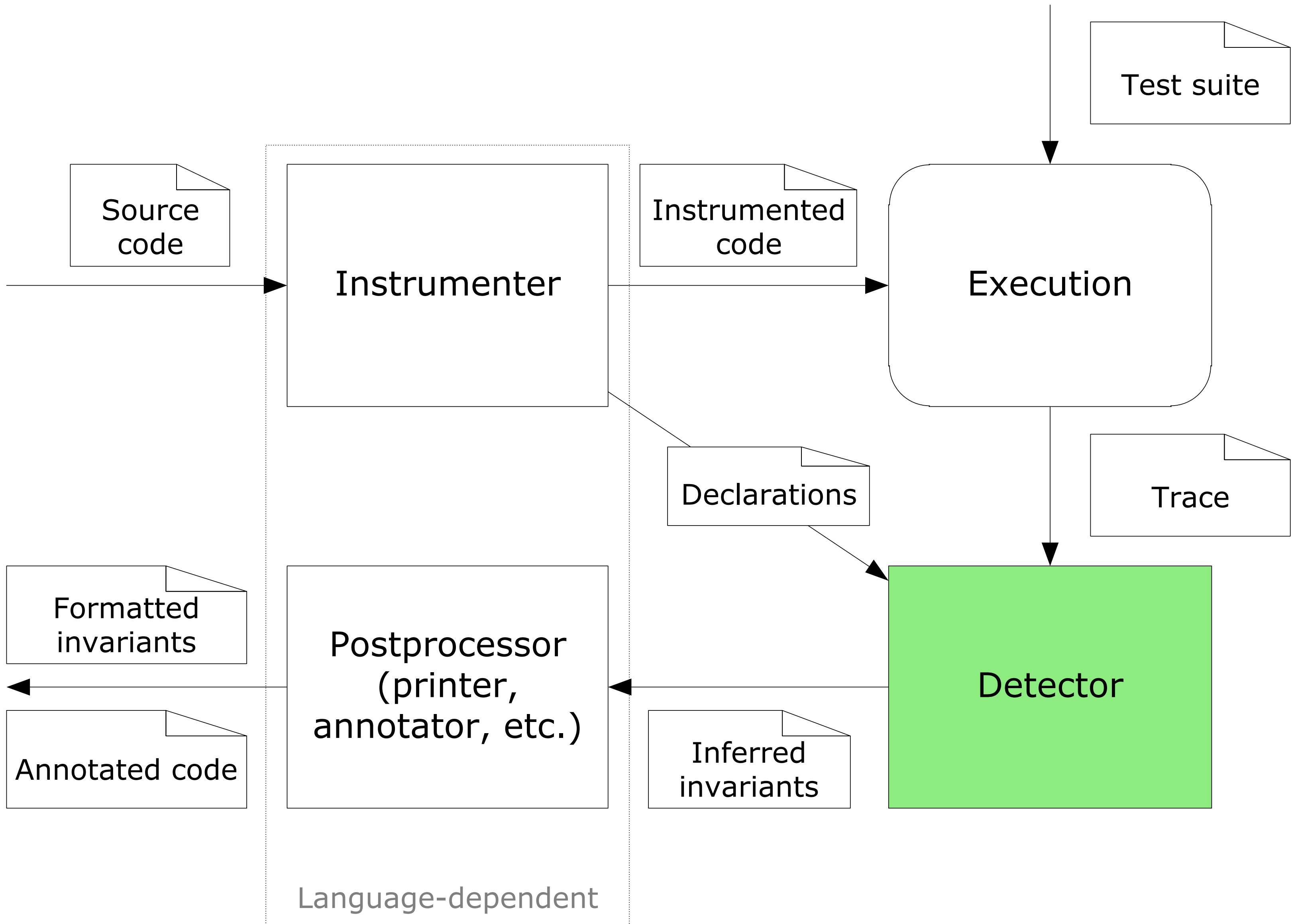
```
class BANK_ACCOUNT
  ...
  balance: INTEGER

  deposit (amount: INTEGER)
      do

●
            balance := balance + amount

●
      end
end
```

Daikon architecture



Detector

- Has a predefined set of invariant templates
- At each program point instantiates the templates with appropriate variables
- Checks invariants against program point samples (variable values in the trace)
- Reports invariants that are not falsified (and satisfy other conditions)

Detector: example

- Templates: *x* = const   *x* >= const   *x* = *y*   ...
- Program point: BANK_ACCOUNT.deposit:::ENTER
- Variables: *balance*, *amount*: INTEGER
- Invariants:

~~*balance* = 0~~

balance >= 0

~~*amount* = 10~~

amount >= 1

~~*balance* = *amount*~~

- Samples:

balance 0 *amount* 10

balance 10 *amount* 20

balance 30 *amount* 1

Unary invariant templates

- Constant

  x = const

- Bounds

  x < const (<=, >, >=)

- Nonzero

  x /= 0

- Modulus

  x = r mod m

- No duplicates

  s has no duplicates

- index and element

  s [i] = i (<, <=, >, >=)

Binary invariant templates

- Comparisons

  $x = y$ (<, <=, >, >=)

- Linear binary

  $ax + by = 0$

- Squared

  $x = y$^2

- Divides

  $x = 0 \text{ mod } y$

- Zero track

  $x = 0$ implies $y = 0$

- Member

  x in s

- Reversed

  $s1 = s2$.reveresed

- Subsequence and subset

  $s1$ is subsequence of $s2$ $s1$ is subset of $s2$

Ternary invariant templates

- Linear ternary

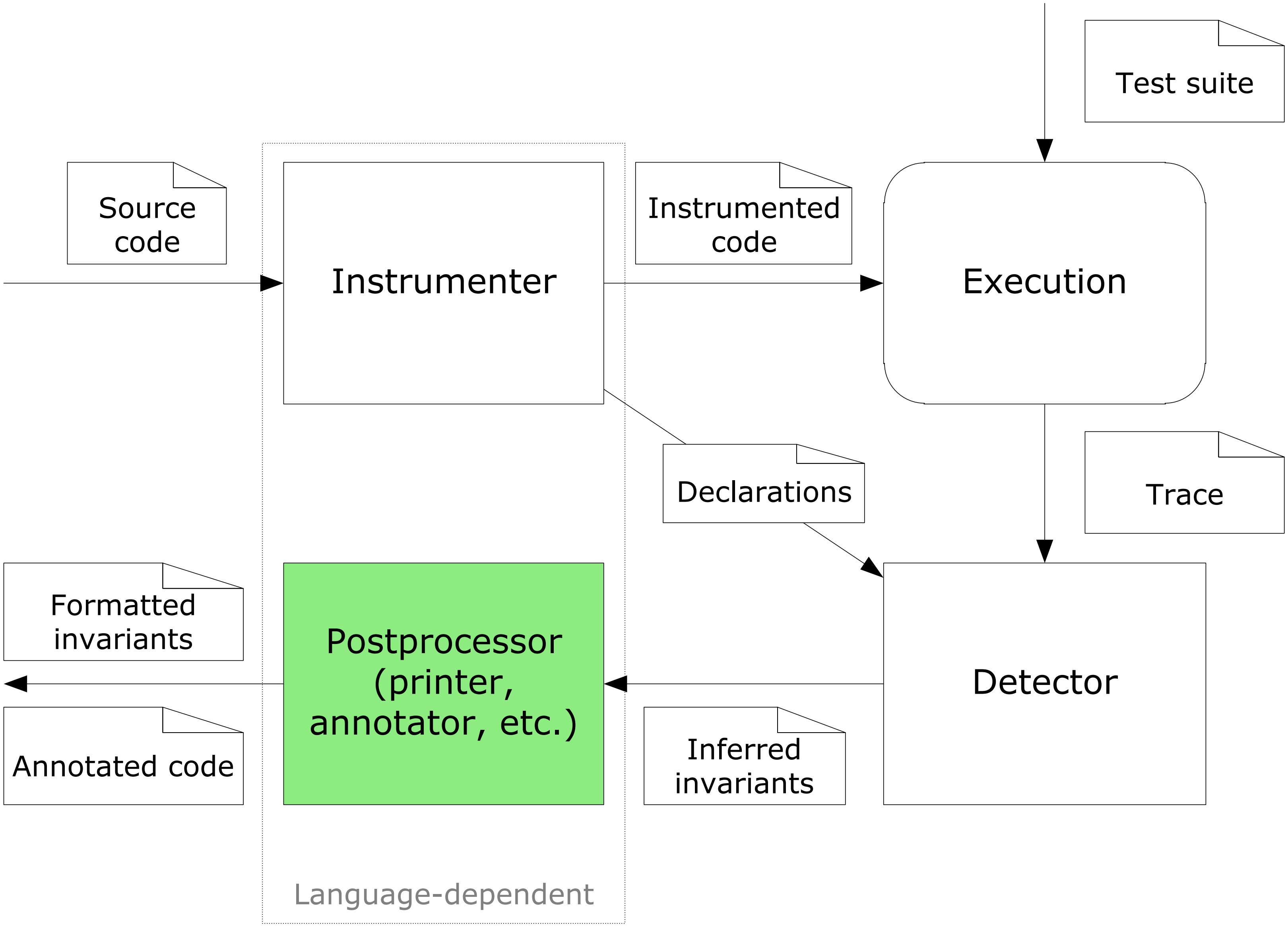
$$ax + by + zc = 0$$

- Binary function

$$z = f\,(x,\ y)$$

where f = and, or, xor, min, max, gcd, pow

Daikon architecture



Annotator

- Annotates code with inferred invariants

```
class BANK_ACCOUNT
    ...
    balance: INTEGER

    deposit (amount: INTEGER)



        do
            balance := balance + amount
        end
end
```

```
BANK_ACCOUNT.deposit:::ENTER
    balance >= 0
    amount >= 1
...
```


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

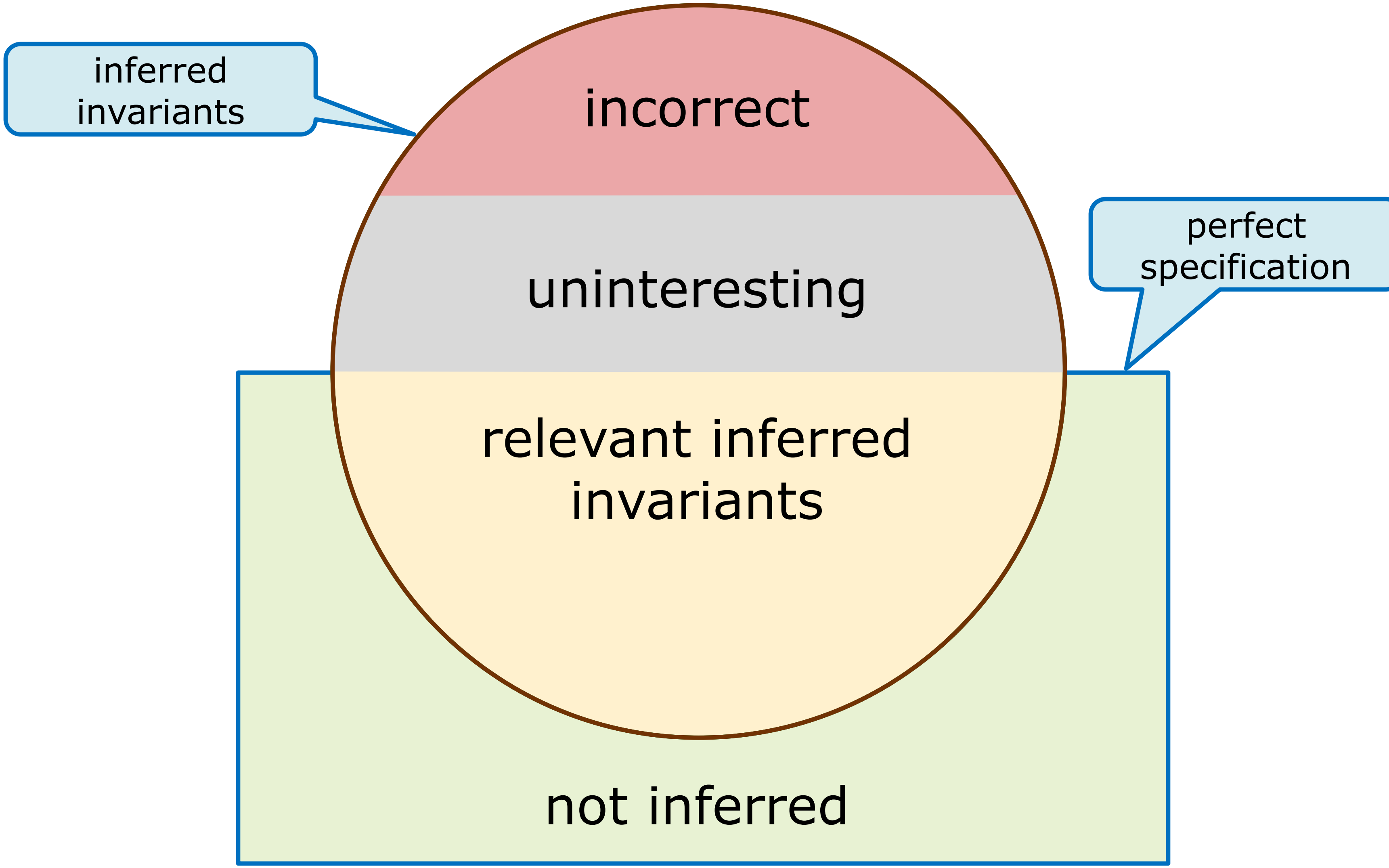
Results depend on...

- Source code
- Invariant templates
- Variables that instrumenter finds
  - potentially all expressions that can be evaluated at a program point
  - needs to choose interesting ones
- Test suite
- Fine tuning the detector

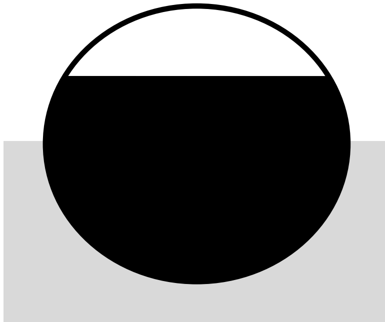
Dynamic inference is...

- Not sound
  - Sound over the test suite, but not potential runs
- Not complete
  - Restricted to the set of templates and variables
  - Heuristics for eliminating irrelevant invariants might remove relevant ones
- Even if it was, it reports properties of the code, not the developers intent

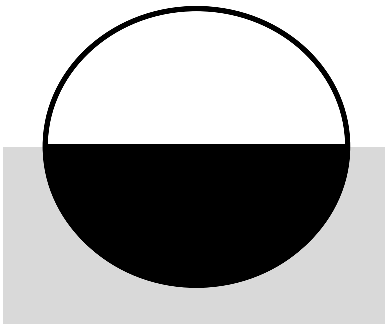
Classification



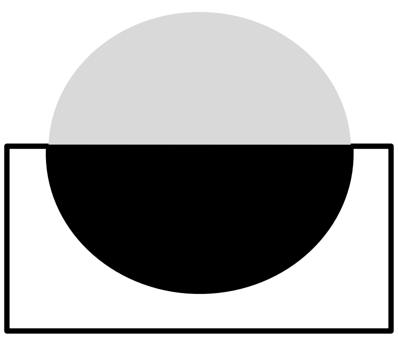
Quality measures



- Correctness – percentage of correct inferred invariants (true code properties)



- Relevance (precision) – percentage of relevant inferred invariants



- Recall – percentage of true invariants that were inferred

Using inferred invariants

- As a specification (after human inspection)
  - Strengthening and correcting human-written specifications
  - Inferring loop invariants that are difficult to construct manually
- Finding bugs
- Evaluating and improving test suites
- Comparing several versions of a program

Improving quality

- Improving relevance
  - Statistical test
  - Redundant invariants
  - Comparability analysis
- Improving recall
  - More templates and variables
  - Conditional invariants

Statistical test

- Checking invariant

$$x \ /= \ 0$$

- Let samples of x be nonzero, distributed in [-5, 5]
  - With 3 samples:

$$p_{by\_chance} = (1 - 1/11)^3 \approx 0.75$$

unjustified

  - With 100 samples:

$$p_{by\_chance} = (1 - 1/11)^{100} \approx 0.00007$$

justified

- Each invariant calculates probability in its own way
- Threshold is defined by the user (usually < 0.01)

Redundant invariants

ensure

$x > 0$

~~x /= 0~~

...

- Invariants that are implied by other invariants are not interesting
- How to find them?
  - General-purpose theorem prover
  - Daikon has built-in hierarchy of invariants (invariants know their suppressors)

Comparability analysis


```
class BANK_ACCOUNT
...
invariant
    number > owner.birth_year
end
```

true, but uninteresting

- Using the same syntactic type (INTEGER) to represent multiple semantic types
- Semantics types can be recovered by static analysis
- Variables x and y are considered comparable if they appear in constructs like

$$x = y \quad x := y \quad x > y \quad x + y \quad \ldots$$

Improving recall

It is easy:

- add more invariant templates
- add more variables of interest

However that increases the search space and

- either makes inference intractable
- or decreases relevance

Choose templates and variables in a smart way

e.g. at the entry to *withdraw* (*amount*: INTEGER) *is_amount_available* (*amount*) is a good choice but *is_amount_available* (5) is not

Conditional invariants

- Invariants of the form

  $$(P_1 \textbf{ and } P_2 \ldots \textbf{ and } P_m) \textbf{ implies } Q$$

  are hard to infer with the basic technique:
  it has to try all combinations of P_i and Q

- An efficient way: Decision Tree Learning

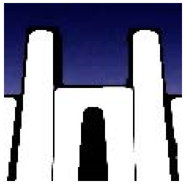
old *after*

False: *index* = **old** *index*

True: *index* = **old** *index* + 1

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CITADEL



- Contract Inference Tool Applying Daikon to Eiffel Language

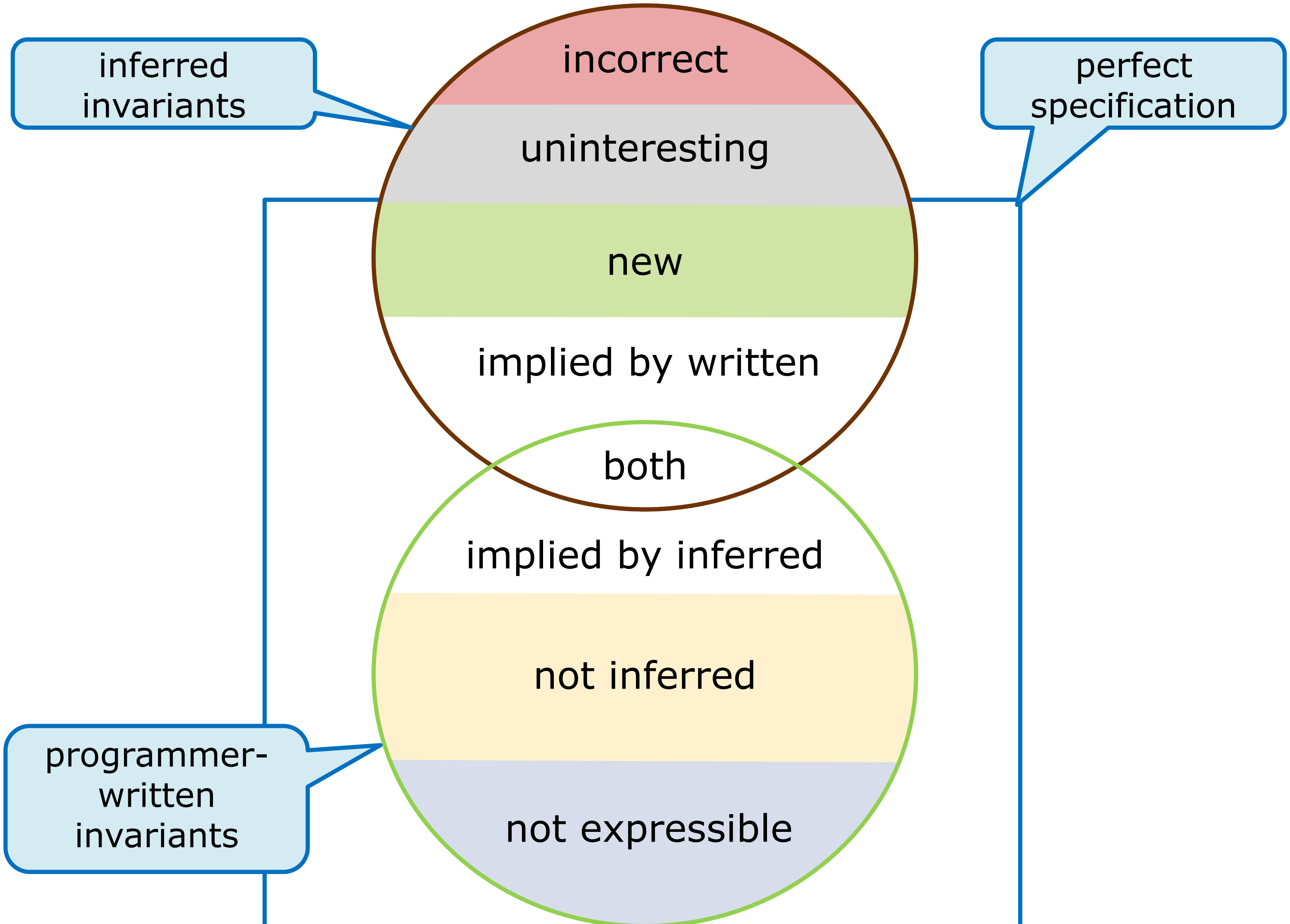
http://se.inf.ethz.ch/people/polikarpova/citadel.html

- Infers only contracts expressible in Eiffel
  - no invariants over sequences
- Uses zero-argument functions as variables
  - Eiffel functions are pure
  - user-supplied preconditions are used to check whether a function can be called
- Infers loop invariants

Experiment

- Comparing programmer-written contracts with inferred ones
- Scope: *25* classes (*89–1501* lines of code)
  - *15* from industrial-grade libraries
  - *4* from an application used in teaching CS at ETH
  - *6* from student projects
- Tests suite: *50* calls to every method, random inputs + partition testing
- Contract clauses total:
  - programmer-written: *831*
  - inferred: *9’349*

Classification



Results

Measure	Description	Value
Correctness	$\frac{\text{correct IC}}{\text{IC}}$	90%
Relevance	$\frac{\text{relevant IC}}{\text{IC}}$	64%
Expressibility	$\frac{\text{PC expressible in Daikon}}{\text{PC}}$	86%
Recall	$\frac{\text{inferred PC}}{\text{PC}}$	59%
Strengthening factor	$\frac{\text{PC + relevant IC}}{\text{PC}}$	5.1

IC = Inferred contract Clauses

PC = Programmer-written contract Clauses

DEMO

AutoInfer

http://se.inf.ethz.ch/research/autoinfer

- Does not use Daikon
- Uses AutoTest to generate the test suite
- Infers universally quantified expressions and implications
- Uses functions with arguments as variables
- Only infers postconditions of commands

Example: LIST.*extend*


```
extend (v: G)
        -- Add `v' to end. Do not move cursor.
    ...
    ensure
        occurrences (v) = occurrences (v) + 1
        count = old count + 1
        i_th (old count + 1) = v
        forall i . 1 <= i <= old count implies i_th (i) = old i_th (i)
        old after implies index = old index + 1
        not old after implies index = old index
        last = v
        forall o:G /= v . occurrences (o) = old occurrences (o)
        forall o:G /=v . has (o) = old has (o)
```