Dynamic Contract Inference
with Daikon and CITADEL

Nadia Polikarpova

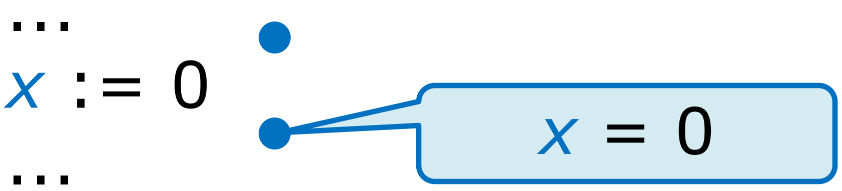
Software Verification

20.10.2010

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
...
```

$x = 0$

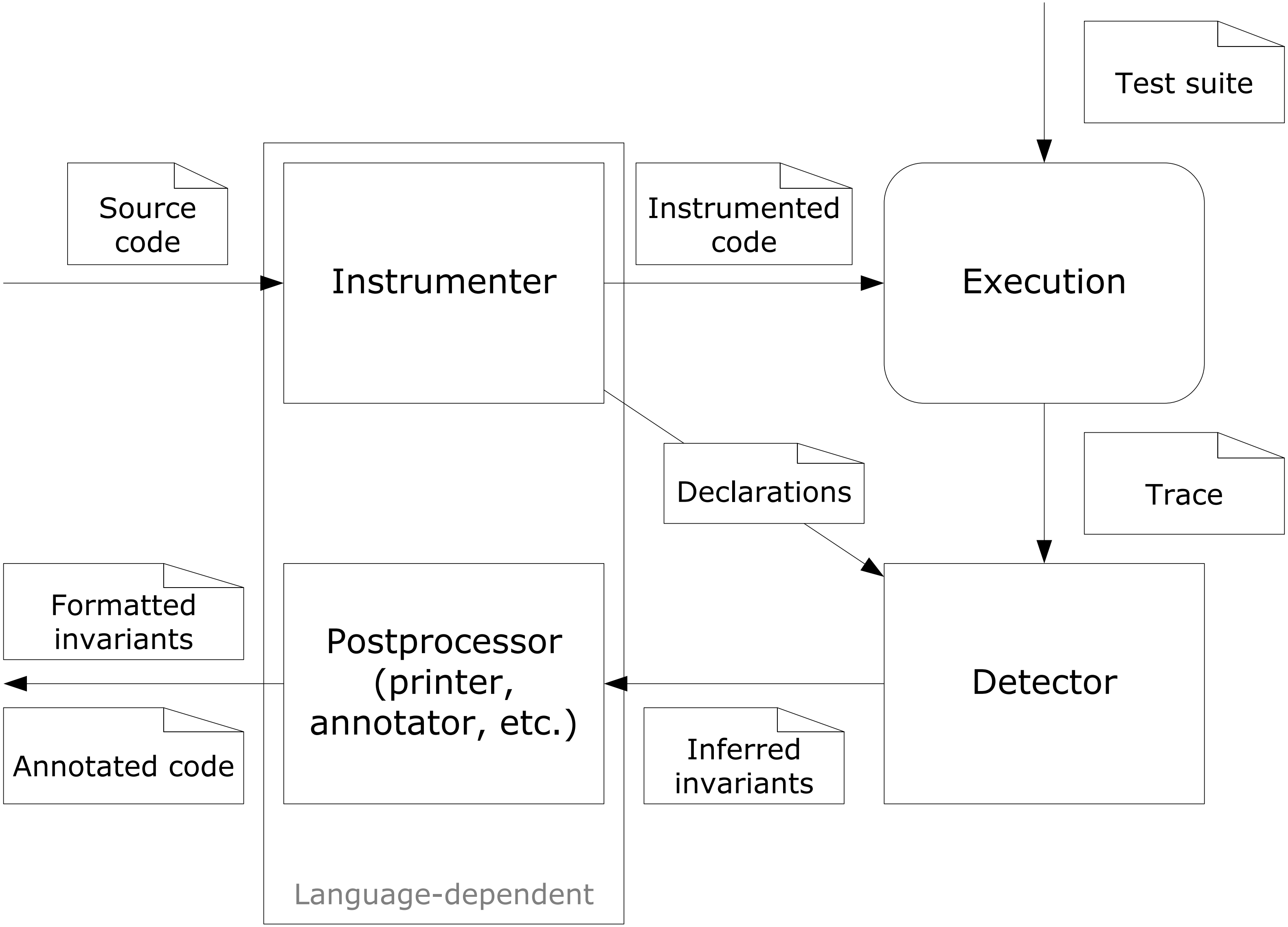
- Dynamic invariant inference – detecting program invariants from values observed during *execution*
- Also called: contract inference, specification inference, assertion inference, ...
- One of the best-known tools is Daikon

  http://groups.csail.mit.edu/pag/daikon/

Overview

- The idea behind Daikon
- Inferred invariants
- Improving inferred invariants
- Contract inference in Eiffel: CITADEL
- A small demo

Daikon architecture



Instrumenter

- Finds program points of interest
  - routine enter/exit, loop enter/exit
- Finds visible variables at these program points
  - current object, formals, locals, return value
- Prints static information about program points and variables
- Modifies the source code so that every time a program point of interest is executed, values of visible variables are printed to the trace file

Instrumenter: example

```
class BANK_ACCOUNT
    …
    balance: INTEGER

    deposit (amount: INTEGER)
        do
            trace.print (“BANK_ACCOUNT.deposit:::ENTER”)
            trace.print (“amount ” + amount.out)
            trace.print (“balance” + balance.out)
            balance := balance + amount
            trace.print (“BANK_ACCOUNT.deposit:::EXIT”)
            trace.print (“amount ” + amount.out)
            trace.print (“balance” + balance.out)
        end
end
```

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

Annotator

- Annotates code with inferred invariants

```
class BANK_ACCOUNT
    ...
    balance: INTEGER

    deposit (amount: INTEGER)
        require
            balance >= 0
            amount >= 1
        do
            balance := balance + amount
        end
end
```

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

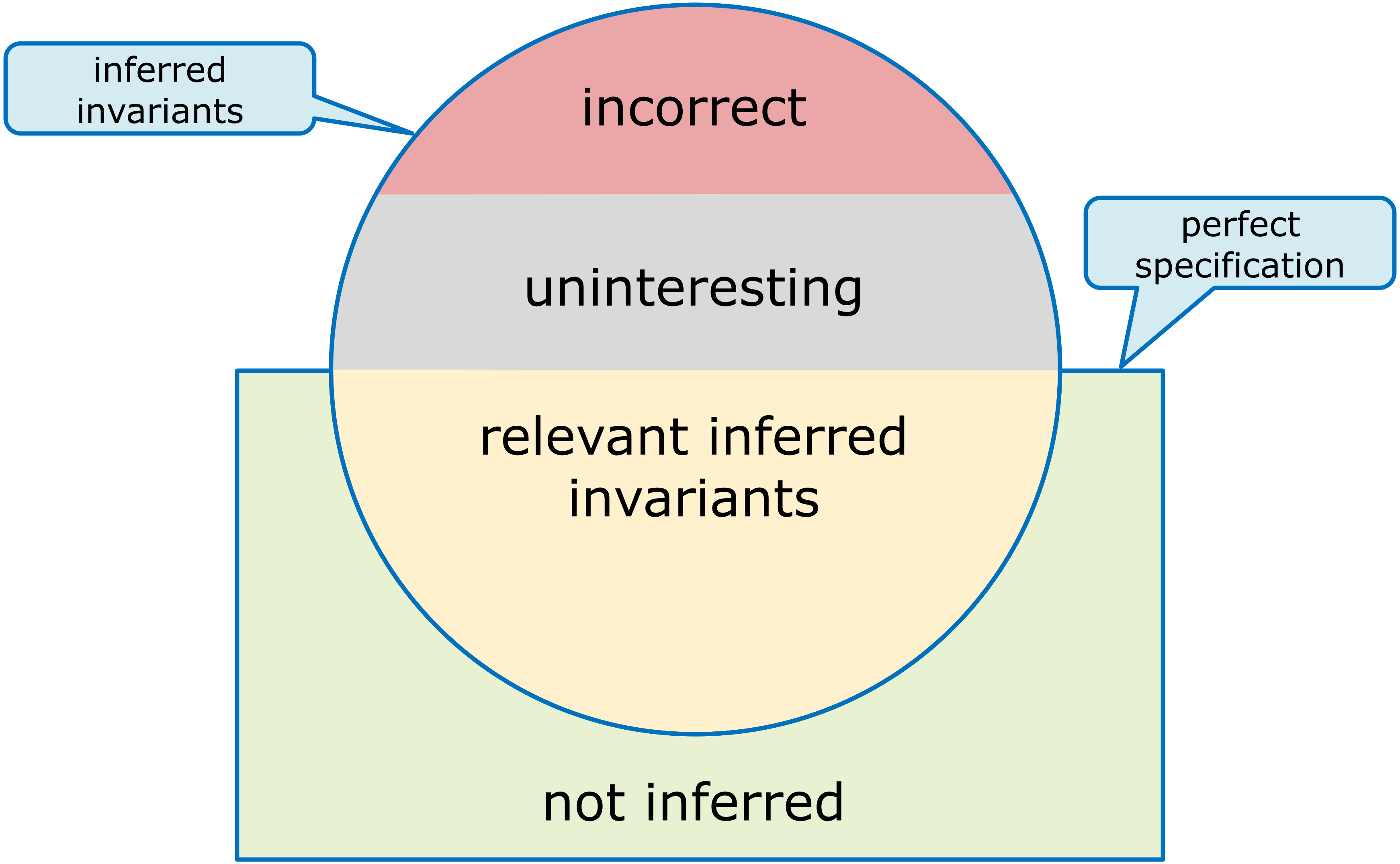
Results depend on...

- Source code
- Invariant templates
- Variables that instrumenter finds
    - potentially infinite set
    - needs to chose interesting ones
- Test suite
- Fine tuning the detector

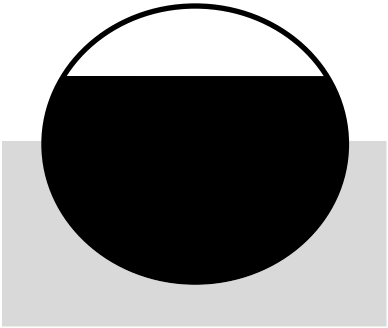
Dynamic inference is...

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

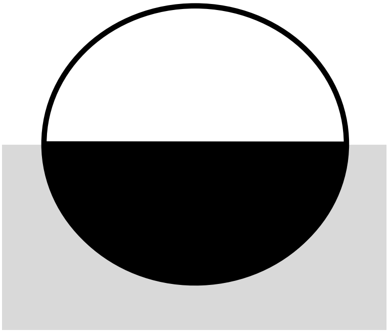
Classification



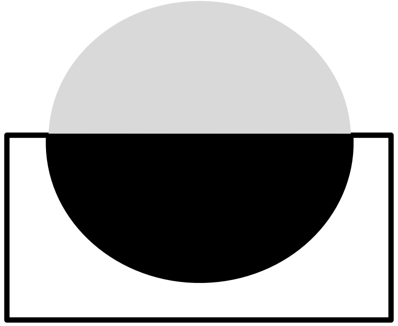
Quality measures



- 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

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

  $x = 0 \bmod y$

- 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

Improving quality

- Improving recall
  - Derived variables
  - Conditional invariants
  - Polymorphism elimination
- Improving relevance
  - Statistical test
  - Redundant invariants
  - Comparability analysis

Derived variables (1)

Variables that instrumenter finds:

- Explicit program entities: current object, formals, locals, return value

  Current = 27656920 *amount* = 10

- Fields of other variables

  Current.*balance* = 10

- Function calls on other variables

  Current.*out* = “Account #1234 balance 10”

- Sequence representation of collections

  Current.*deposits* [] = [10, 20, 100]

- Any other properties

  Current.*deposits* [].is_cyclic = **False**

Derived variables (2)

Variables added by the detector:

- Unary

  s [1] s.length s.max s.min s.sum

- Binary

  s [x] s [1 .. x] s [x .. s.length]

  $s1$.union ($s2$) $s1$.intersect ($s2$) $s1$ ++ $s2$

- Ternary

  s [x .. y]

Conditional invariants

- Invariants of the form

  Q **or** *R*   *P* **implies** *Q*   **if** *P* **then** *Q* **else** *R*

- Technique
  - split the set of samples (with a predicate *P*)
  - infer invariants separately over subsets
- How to split?
  - Static analysis
  - Special values
  - Programmer-directed
  - Exceptions to detected invariants
  - Random split

Polymorphism elimination

- Useful in languages without genericity

```
l: LIST -- List of ANY
...
l.put (1); l.put (2); l.put (3)
```

l [] > 0

- First pass
  - monitor runtime types of variables
  - detect variables with stable runtime type
- Second pass
  - instrumenter processes these variable as if they have the runtime type

Statistical test

- Checking invariant

$$x \mathrel{/=} 0$$

- 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

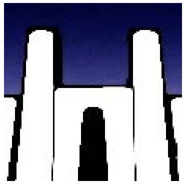
```
class BANK_ACCOUNT
...
invariant
    number > owner.birth_year
end
```

true, but nonsensical

- 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*   *x* := *y*   *x* > *y*   *x* + *y*   ...

CITADEL



- Contract Inference Tool Applying Daikon to Eiffel Language

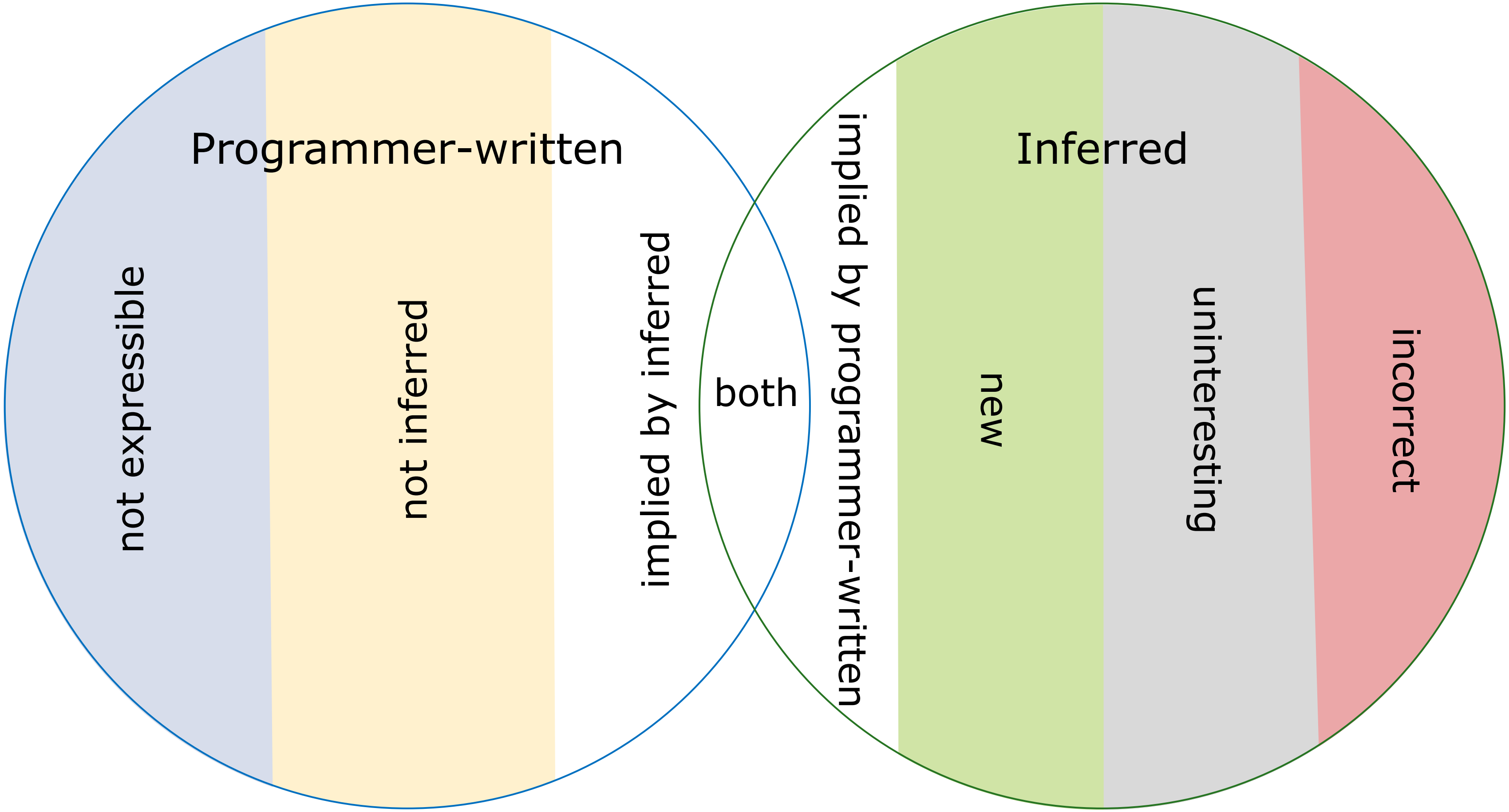
http://se.inf.ethz.ch/people/polikarpova/citadel.html

- 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