Abstract interpretation

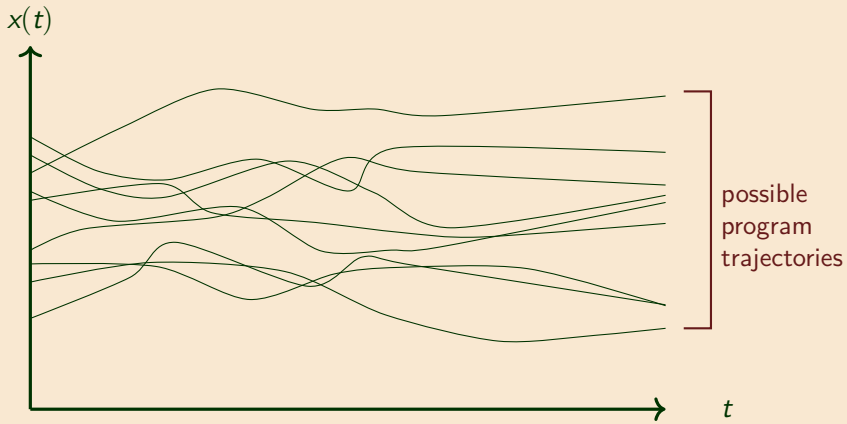
Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

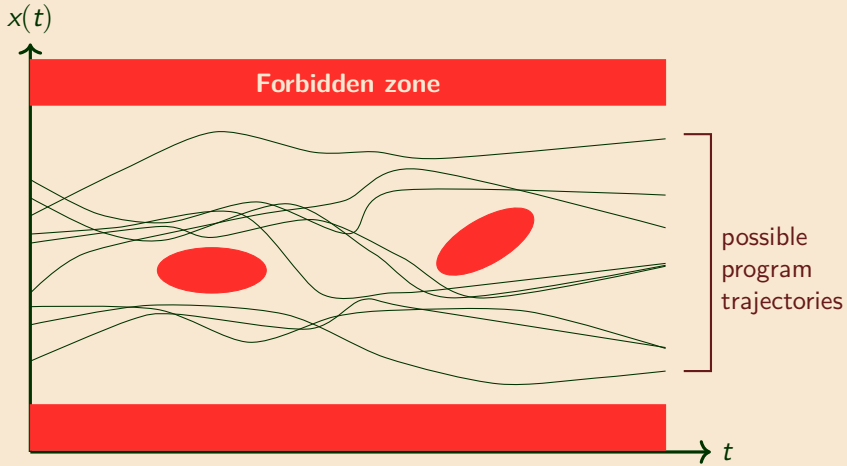
Overview[1]

[1] Based on Patrick Cousot's *Abstract Interpretation in a Nutshell*

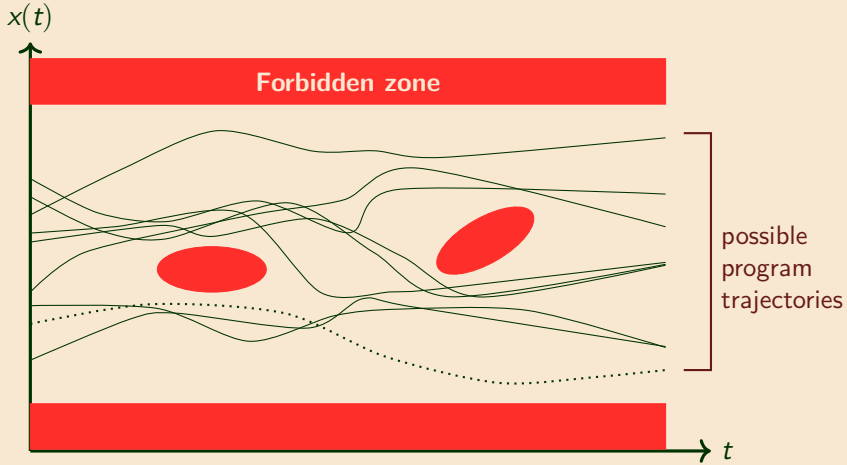
Possible program executions



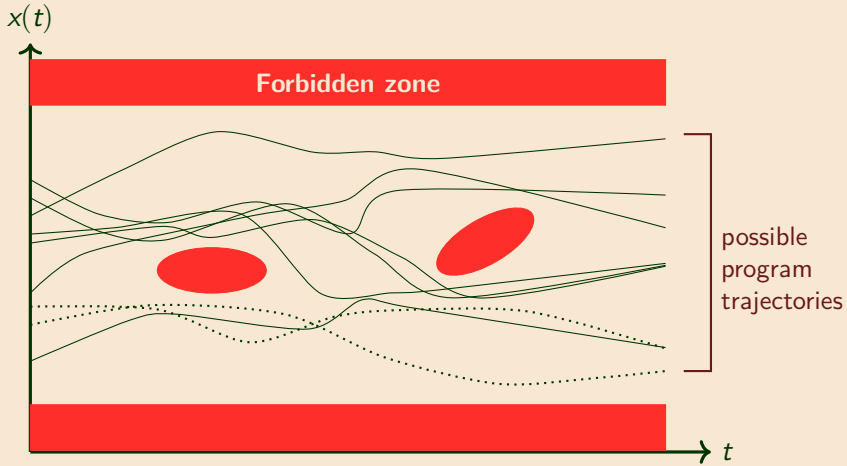
Testing cannot (generally) ensure complete absence of errors.



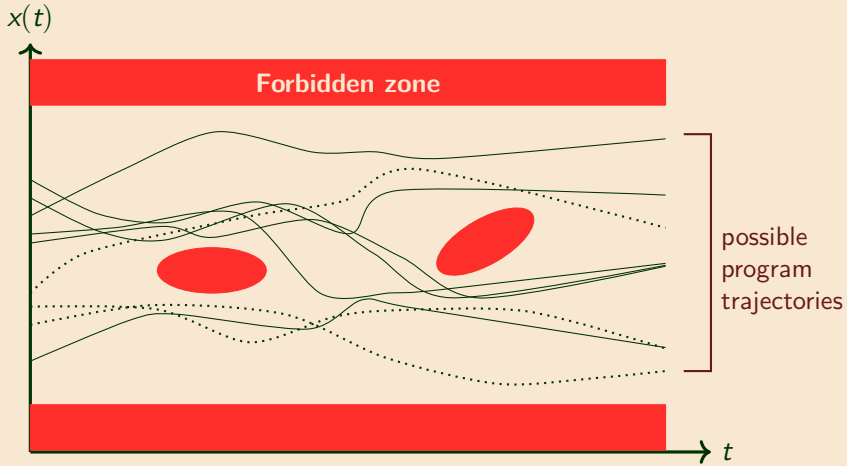
Testing cannot (generally) ensure complete absence of errors.



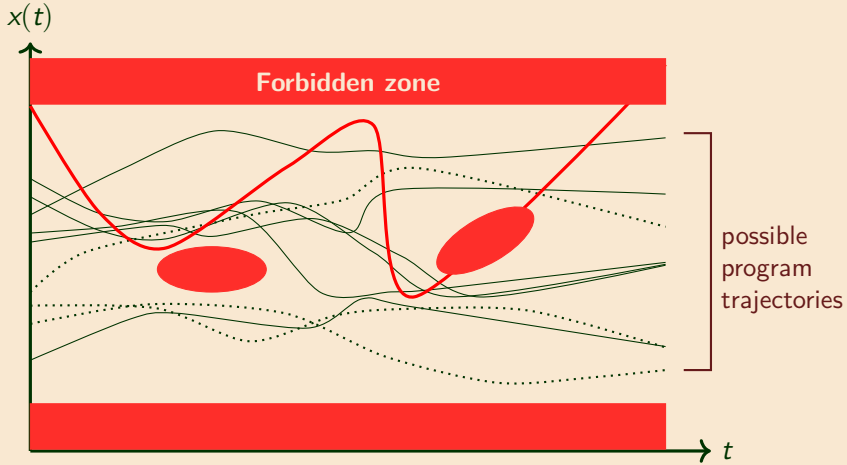
Testing cannot (generally) ensure complete absence of errors.



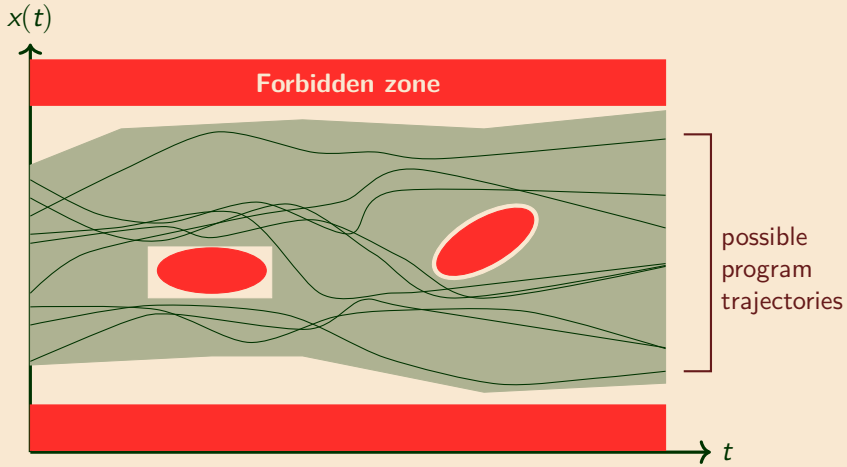
Testing cannot (generally) ensure complete absence of errors.



Testing cannot (generally) ensure complete absence of errors.



Idea: over-approximate all traces to **ensure absence of errors**.



The AI recipe[2]

[2] Adapted from Isil Dillig's *Abstract Interpretation* slides

1. An **abstract domain** that captures some aspect of program invariants
   (e.g. $n \leq x \leq m$ (x always lies within some *interval*))

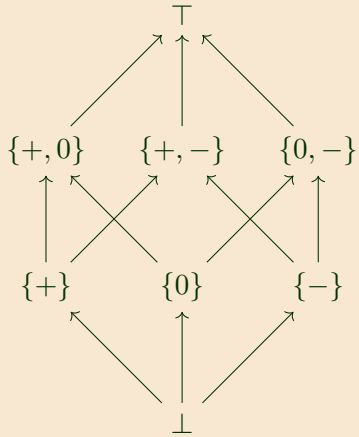
2. An **abstract semantics** that symbolically interprets each program construct
   (e.g. given invariants on x and y, what are the invariants on $x + y$?)

3. Iterate until **fixed point**

Example: *sign* abstract domain

Functions: **concretization** (γ) and **abstraction** (α) map between abstract values & sets of concrete values:

$$\gamma(\{+,-\}) = \{x \in \mathbb{Z} \mid x \neq 0\}$$
$$\dots$$
$$\alpha(\{-1,-2,4\}) = \{+,-\}$$
$$\dots$$



Abstract semantics for $+$

$+$	$\bot$	$\{+\}$	$\{0\}$	$\{-\}$	$\{+,0\}$	$\{+,-\}$	$\dots$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\dots$
$\{+\}$	$\bot$	$\{+\}$	$\{+\}$	$\top$	$\{+\}$	$\top$	$\dots$
$\{0\}$	$\bot$	$\{+\}$	$\{0\}$	$\{-\}$	$\{+,0\}$	$\{+,-\}$	$\dots$
$\{-\}$	$\bot$	$\top$	$\{-\}$	$\{-\}$	$\top$	$\top$	$\dots$
$\{+,0\}$	$\bot$	$\{+\}$	$\{+,0\}$	$\top$	$\{+,0\}$	$\top$	$\dots$
$\{+,-\}$	$\bot$	$\top$	$\{+,-\}$	$\top$	$\top$	$\top$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

```
int x = 2;
int y = 0;
while (y != z) {
   if (f y) {x = x + 1;}
   y = y + x
}
/* What do we know
 about x and y here? */
```

Evolution of x and y:

	x	y
0	{+}	{0}
1	{+}	{+,0}
2	{+}	{+,0}

(Generally: fixed point calculation may not terminate; we may need **widening**.)

Reading

Paper 1: Octagons

The Octagon Abstract Domain

Antoine Miné

École Normale Supérieure de Paris, France,
mine@di.ens.fr,
http://www.di.ens.fr/~mine

Abstract—This article presents a new numerical abstract domain for static analysis by abstract interpretation. It extends a former numerical abstract domain based on Difference-Bound Matrices and allows us to represent invariants of the form $(\pm x \pm y \leq c)$, where x and y are program variables and c is a real constant.

We focus on giving an efficient representation based on Difference-Bound Matrices, with, for n variables, $O(n^2)$ memory use, the number of variables—and graph-based algorithms for all common abstract operators—$O(n^3)$ time cost. This includes a normal form algorithm in test equivalence of representation and a widening operator to compute least fixpoint approximations.

Index Terms—abstract interpretation, abstract domains, linear invariants, safety analysis, static analysis tools.

I. Introduction

This article presents practical algorithms to represent and manipulate invariants of the form $(\pm x \pm y \leq c)$, where x and y are numerical variables and c is a numeric constant. It extends the analysis we previously proposed in our PADO-II article [1]. Sets described by such invariants are special kind of polyhedra called octagons because they feature at most eight edges in dimension 2 (Figure 2). Using abstract interpretation, this allows discovering automatically common errors, such as division by zero, out-of-bound array access or deadlock, and more generally to prove safety properties for programs.

Our method works well for reals and rationals. Integer variables can be assumed, in the analysis, to be real in order to find approximate but safe invariants.

Example. The very simple program described in Figure 1 simulates M one-dimensional random walks of m steps and stores the bits in the array *tab*. Assertions in curly braces are discovered automatically by a simple static analysis using our octagonal abstract domain. Thanks to the invariants discovered, we have the guarantee that the program does not perform out-of-bound array access at lines 2 and 10. The difficult point in this example is the fact that the bounds of the array *tab* are not known at the time of the analysis; thus, they must be treated symbolically.

For the sake of brevity, we omit proofs of theorems in this article. The complete proof for all theorems can be found in the author's Master thesis [2].

```
1   int tab[−m .. m];
2   for i = −m to m                    tab[i] = 0;   {−m ≤ i ≤ m}
3   for j = 1 to M do
4       int a = 0;
5       for i = 1 to m
6           { 1 ≤ i ≤ m; −i + 1 ≤ a ≤ i − 1 }
7           if rand(2) = 0
8               then a = a + 1;   {−i + 1 ≤ a ≤ i}
9               else a = a − 1;   {−i ≤ a ≤ i − 1}
10          tab[a] = tab[a] + 1;   {−m ≤ a ≤ m}
11  done.
```

Fig. 1. Simulation of a random walk. The assertions in curly brackets { . } are discovered automatically and prove that this program does not perform index out of bound error when accessing the array *tab*.

II. Previous Work

A. Numerical Abstract Domains.

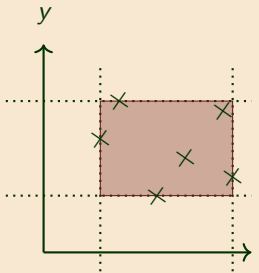
Static analysis has developed a successful methodology, based on the abstract interpretation framework—see Cousot and Cousot's POPL'77 paper [3]—to build analyzers that discover invariants automatically: all we need is an *abstract domain*, which is a practical representation of the invariants we want to study, together with a fixed set of operators and transfer functions (union, intersection, widening, assignment, guard, etc.) as described in Cousot and Cousot's POPL'79 article [4].

There exists many *numerical abstract domains*. The most used are the lattice of *intervals* (described in Cousot and Cousot's ESOP'76 article [5]) and the lattice of *polyhedra* (described in Cousot and Halbwachs's POPL'78 article [6]). They represent, respectively, invariants of the form $(v \in [c_1, c_2])$ and $(v_1 c_1 + \cdots + v_n c_n \leq c)$, where $v_1 v_2, \ldots, v_n$ are program variables and $c, c_1, c_2, c_1, \ldots, c_n$ are constants. Whereas the interval analysis is very efficient—linear memory and time cost—but not very precise, the polyhedron analysis is much more precise (Figure 2) but has a huge memory cost—in practice, it is exponential in the number of variables.

Remark that the correctness of the program in Figure 1 depends on the discovery of invariants of the form $(a \in [-m, m])$ where m must not be treated as a constant, but as a *variable*—its value is not known at analysis time. Thus, this example is beyond the scope of interval analysis. It can be solved, of course, using polyhedron analysis.

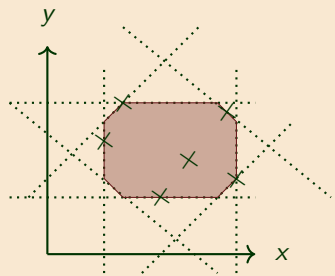
B. Difference-Bound Matrices.

Several satisfiability algorithms for set of constraints involving only two variables per constraint have been proposed in order to solve *Constraint Logic Programming (CLP) problems*. Pratt analyses, in [7], the simple case of constraints of the



Interval domain

$a \leq x \leq b,\ c \leq y \leq d$



Octagon domain

$\pm x \pm y < c$

Fast Polyhedra Abstract Domain

Gagandeep Singh Markus Püschel Martin Vechev
Department of Computer Science
ETH Zurich, Switzerland
{gsingh, pueschel, martin.vechev}@inf.ethz.ch

Abstract

Numerical abstract domains are an important ingredient of modern static analyzers used for verifying critical program properties (e.g., absence of buffer overflow or memory safety). Among the many numerical domains introduced over the years, Polyhedra is the most expressive one, but also the most expensive: it has worst-case exponential space and time complexity. As a consequence, static analysis with the Polyhedra domain is thought to be impractical when applied to large scale, real world programs.

In this paper, we present a new approach and a complete implementation for speeding up Polyhedra domain analysis. Our approach does not lose precision, and for many practical cases, is orders of magnitude faster than state-of-the-art solutions. The key insight underlying our work is that polyhedra arising during analysis can usually be kept decomposed, thus considerably reducing the overall complexity.

We first present the theory underlying our approach, which identifies the interaction between partitions of variables and domain operators. Based on the theory we develop new algorithms for these operators that work with decomposed polyhedra. We implemented these algorithms using the same interface as existing libraries, thus enabling static analyzers to use our implementation with little effort. In our evaluation, we analyze large benchmarks from the popular software verification competition, including Linux device drivers with over 50K lines of code. Our experimental results demonstrate massive gains in both space and time: we show end-to-end speedups of two to five orders of magnitude compared to state-of-the-art Polyhedra implementations as well as significant memory gains, on all larger benchmarks. In fact, in many cases our analysis terminates in seconds where prior code runs out of memory or times out after 4 hours.

We believe this work is an important step in making the Polyhedra abstract domain both feasible and practically usable for handling large, real-world programs.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.2.1 [*Numerical Algorithms and Problems*]: Computations on matrices

General Terms Verification, Performance

Keywords Numerical program analysis, Abstract interpretation, Partitions, Polyhedra decomposition, Performance optimization

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01…$15.00
http://dx.doi.org/10.1145/3009837.3009885

1. Introduction

Abstract interpretation is a general framework for static program analysis that defines sound and precise abstractions for the program's (potentially infinite) concrete semantics. The abstract semantics of a program require an *abstract domain* for capturing the properties of interest. Examples of useful program properties involve the program's heap [22, 26], numerical values [6, 14, 16, 17, 19, 30], termination [27, 28], and many others. An abstract interpreter is obtained by defining the effect of statements and expressions in the programming language on the abstract domain. In this paper, we focus on numerical abstract domains which capture numerical relationships between program variables. These relationships are important for proving the absence of buffer overflow, division by zero, and other properties. Thus, numerical domains are an important ingredient of modern static analyzers [4, 9].

Expressivity vs. cost In an ideal setting, one would simply use the most expressive domain to analyze a program, i.e., Polyhedra [6]. However, the Polyhedra domain comes with a worst case exponential complexity in both space and time. Thus, an analyzer using Polyhedra can easily fail to analyze large programs by running out of memory or by timing out. Because of this, the Polyhedra domain is often thought to be impractical, and thus, over the years, researchers have designed domains that limit its expressivity in exchange for better asymptotic complexity. Examples include Octagon [30], Zone [17], Pentagon [16], SubPolyhedra [14] and Gauge [30]. Unfortunately, limited expressivity can make the over-approximation too imprecise for proving the desired property.

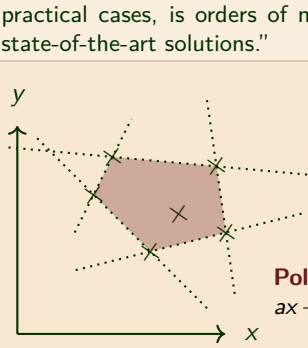
Our work In this work, we revisit the basic assumption that Polyhedra is impractical for static analysis. We present a new approach which enables the application of Polyhedra to large, realistic programs, with speedups ranging between two to five orders of magnitude compared to the state-of-the-art. We note that our approach does not lose precision yet can analyze programs beyond the reach of current approaches.

The key insight of our approach is that the set of program variables partitions into subsets such that linear constraints only exist between variables in the same subset [10, 23]. We leverage this observation to decompose a large polyhedron into a set of smaller polyhedra, thus reducing the asymptotic complexity of the Polyhedra domain. However, maintaining decomposition online is challenging because over 40 Polyhedra operators change the partitions dynamically and in non-trivial ways: subsets can merge, split, grow, or shrink during analysis. Note that an exact partition cannot be computed a priori as otherwise the approach loses precision [4].

To ensure our method does not lose precision, we develop a theoretical framework that asserts how partitions are modified during analysis. We then use this theory to design new abstract operators for Polyhedra. Interestingly, our framework can be used for decomposing other numerical domains, not only Polyhedra.

"Among the many numerical domains introduced over the years, Polyhedra is the most expressive one, but also the most expensive: it has worst-case exponential space and time complexity.

Our approach does not lose precision, and for many practical cases, is orders of magnitude faster than state-of-the-art solutions."



Polyhedra domain

$ax + by < c$

A Formally-Verified C Static Analyzer

Jacques-Henri Jourdan
Inria Paris-Rocquencourt
jacques-henri.jourdan@inria.fr

Vincent Laporte
IRISA and U. Rennes 1
vincent.laporte@irisa.fr

Sandrine Blazy
IRISA and U. Rennes 1
sandrine.blazy@irisa.fr

Xavier Leroy
Inria Paris-Rocquencourt
xavier.leroy@inria.fr

David Pichardie
IRISA and ENS Rennes
david.pichardie@irisa.fr

Abstract

This paper reports on the design and soundness proof, using the Coq proof assistant, of Verasco, a static analyzer based on abstract interpretation for most of the ISO C 1999 language (excluding recursion and dynamic allocation). Verasco establishes the absence of run-time errors in the analyzed programs. It enjoys a modular architecture that supports the extensible combination of multiple abstract domains, both relational and non-relational. Verasco integrates with the CompCert formally-verified C compiler so that not only the soundness of the analysis results is guaranteed with mathematical certitude, but also the fact that these guarantees carry over to the compiled code.

Categories and Subject Descriptors D.2.4 [*Software Engineering*] Software/Program Verification—*Assertion checkers, Correctness proofs*; F.3.1 [*Logics and meanings of programs*] Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

Keywords static analysis; abstract interpretation; soundness proofs; proof assistants

1. Introduction

Verification tools are increasingly used during the development and validation of critical software. These tools provide guarantees that are always independent from those obtained by more conventional means such as testing and code review; often stronger; and sometimes cheaper to obtain (rigorous testing can be very expensive). Verification tools are based on a variety of techniques such as static analysis, model checking, deductive program proof, and combinations thereof. The guarantees they provide range from basic memory safety to full functional correctness. In this paper, we focus on static analyzers for low-level, C-like languages that establish the absence of run-time errors such as out-of-bound array accesses, null pointer dereference, and arithmetic exceptions. These basic properties are essential both for safety and security. Among the various verification techniques, static analysis is perhaps the one that scales best to large existing code bases, with minimal intervention from the programmer.

Static analyzers can be used in two different ways: as sophisticated bug finders, discovering potential programming errors that are hard to find by testing; or as specialized program verifiers, establishing that a given safety or security property holds with high confidence. For bug-finding, the analysis must be precise (too many false alarms render the tool unusable for this purpose), but no guarantee is offered nor expected that all bugs of a certain class will be found. For program verification, in contrast, *soundness* of the analysis is paramount: if the analyzer reports no alarms, it must be the case that the program is free of the class of run-time errors tracked by the analyzer; in particular, all possible execution paths through the program must be accounted for.

To use a static analyzer as a verification tool, and obtain certification credit in regulations such as DO-178C (avionics) or Common Criteria (security), evidence of soundness of the analyzer must therefore be provided. Owing to the complexity of static analyzers and of their input data (programs written in "big" programming languages), rigorous testing of a static analyzer is very difficult. Even if the analyzer is built on mathematically-rigorous grounds such as abstract interpretation [14], the possibility of an implementation bug remains. The alternative we investigate in this paper is to build the analyzer and prove its soundness, using a tool for program proof, mechanized with the Coq proof assistant, to check the implementation of a static analyzer in order to prove its soundness with respect to the dynamic semantics of the analyzed language.

Our analyzer, called Verasco, is based on abstract interpretation; handles most of the ISO C 1999 language, with the exception of recursion and dynamic memory allocation; combines several abstract domains, both non-relational (integer intervals and congruences, floating-point intervals, points-to sets) and relational (convex polyhedra, symbolic equalities); and is entirely proved to be sound using the Coq proof assistant. Moreover, Verasco is connected to the CompCert C formally-verified compiler [26], ensuring that the safety guarantees established by Verasco carry over to the compiled code.

Mechanizing soundness proofs of verification tools is not a new idea. It has been applied at large scale to Java type-checking and bytecode verification [25], proof-carrying code infrastructures [1, 12], and verification condition generators for C-like languages [20, 23], among other projects. The formal verification of static analyzers based on dataflow analysis or abstract interpretation is less developed. As detailed in section 10, earlier work in this area either

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00
http://dx.doi.org/10.1145/2676726.2676964

"Verasco, a static analyzer based on abstract interpretation for most of the ISO C 1999 language (excluding recursion and dynamic allocation).

"Verasco establishes the absence of run-time errors in the analyzed programs. It enjoys a modular architecture that supports the extensible combination of multiple abstract domains, both relational and non-relational."

Abstract interpretation vs types

What are the relative benefits of AI and types?
(Are they in some sense the same thing?)

Cost vs precision

What is the tradeoff?

Widening and narrowing

What role do they play in convergence and precision?

Applicability

How widely applicable is abstract interpretation? How well does it scale up?

Relational and non-relational domains