Abstract interpretation

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk
Overview\textsuperscript{1}

\textsuperscript{1}Based on Patrick Cousot’s \textit{Abstract Interpretation in a Nutshell}
Possible program executions

Overview

Recipe

Reading

\[ x(t) \]

possible program trajectories
Testing cannot (generally) ensure complete absence of errors.
Testing cannot (generally) ensure complete absence of errors.

\[ x(t) \]

Forbidden zone

possible program trajectories

\[ t \]
Testing cannot (generally) ensure complete absence of errors.

\[ x(t) \]

Forbidden zone

possible program trajectories
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**.

$x(t)$

Forbidden zone

possible program trajectories
The AI recipe

\[^2\text{Adapted from Isil Dillig's } Abstract Interpretation \text{ slides}\]
Three-part recipe

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 ($\gamma$) and abstraction ($\alpha$)
map between abstract values & sets of concrete values:

$$
\gamma(\{+, -\}) = \{x \in \mathbb{Z} \mid x \neq 0\}
\ldots
$$

$$
\alpha(\{-1, -2, 4\}) = \{+, -\}
\ldots
$$
### Abstract semantics for $+$

<table>
<thead>
<tr>
<th></th>
<th>$+$</th>
<th>$\perp$</th>
<th>${+}$</th>
<th>${0}$</th>
<th>${-}$</th>
<th>${+, 0}$</th>
<th>${+, -, }$</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>${+}$</td>
<td>$\perp$</td>
<td>${+}$</td>
<td>${+}$</td>
<td>$\top$</td>
<td>${+}$</td>
<td>$\top$</td>
<td>$\ldots$</td>
<td></td>
</tr>
<tr>
<td>${0}$</td>
<td>$\perp$</td>
<td>${+}$</td>
<td>${0}$</td>
<td>${-}$</td>
<td>${+, 0}$</td>
<td>${+, -, }$</td>
<td>$\ldots$</td>
<td></td>
</tr>
<tr>
<td>${-}$</td>
<td>$\perp$</td>
<td>$\top$</td>
<td>${-}$</td>
<td>${-}$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>${+, 0}$</td>
<td>$\perp$</td>
<td>${+}$</td>
<td>${+, 0}$</td>
<td>$\top$</td>
<td>${+, 0}$</td>
<td>$\top$</td>
<td>$\ldots$</td>
<td></td>
</tr>
<tr>
<td>${+, -, }$</td>
<td>$\perp$</td>
<td>$\top$</td>
<td>${+, -, }$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

...
```c
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:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{+}</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td>{+}</td>
<td>{+,0}</td>
</tr>
<tr>
<td>2</td>
<td>{+}</td>
<td>{+,0}</td>
</tr>
</tbody>
</table>

(Generally: fixed point calculation may not terminate; we may need **widening**.)
Reading
The Octagon Abstract Domain

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

Abstract—This article presents a new numerical abstract domain for data analysis for abstract interpretation. It extends a former numerical abstract domain based on Difference-Bound Matrices and allows us to perform invariants of the form $y = (x - a) \times c$ as it is symmetrical to $x = (y - b) / c$.

The octagon domain is characterized by the following properties:

- $a \leq x \leq b$
- $c \leq y \leq c$

This domain is used to find approximate but safe invariants. It is a former numerical abstract domain based on Difference-Bound Matrices and allows us to represent invariants of the form $y = (x - a) \times c$ as it simulates division by zero, out-of-bound array access or deadlock, and this allows discovering automatically common errors, such as index out of bound error when accessing the array $\text{tab}[i]$.Assertions in curly brackets $\{\cdot\}$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The hits are done as a variable $\text{tab}[i]$ is a numeric constant. It is used to check for indexes in array access.

Our method works well for reals and rationals. Integer variables can be assumed, in the analysis, to be real in order to solve Constraint Logic Programming (CLP) problems. 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 form $x \leq y \leq c$ as a variable $x$ is a numeric constant. It is used to check for indexes in array access.

The very simple program described in Figure 1 simulates a random walk. The assertions in curly brackets $\{\cdot\}$ are numerical variables and $\{\cdot\}$ are program variables and $\{\cdot\}$ are not known at the time of the analysis; thus, they must be treated symbolically.

The points $\bullet$ discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The complete proof for all theorems can be found in the author’s Master thesis [2].

The Octagon domain $\pm x + \pm y < c$ is a former numerical abstract domain based on Difference-Bound Matrices and allows us to represent invariants of the form $y = (x - a) \times c$. It simulates division by zero, out-of-bound array access or deadlock, and this allows discovering automatically common errors, such as index out of bound error when accessing the array $\text{tab}[i]$.Assertions in curly brackets $\{\cdot\}$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The hits are done as a variable $\text{tab}[i]$ is a numeric constant. It is used to check for indexes in array access.

Our method works well for reals and rationals. Integer variables can be assumed, in the analysis, to be real in order to solve Constraint Logic Programming (CLP) problems. 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 form $x \leq y \leq c$ as a variable $x$ is a numeric constant. It is used to check for indexes in array access.

The points $\bullet$ discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The complete proof for all theorems can be found in the author’s Master thesis [2].

The Octagon domain $\pm x + \pm y < c$ is a former numerical abstract domain based on Difference-Bound Matrices and allows us to represent invariants of the form $y = (x - a) \times c$. It simulates division by zero, out-of-bound array access or deadlock, and this allows discovering automatically common errors, such as index out of bound error when accessing the array $\text{tab}[i]$.Assertions in curly brackets $\{\cdot\}$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The hits are done as a variable $\text{tab}[i]$ is a numeric constant. It is used to check for indexes in array access.

Our method works well for reals and rationals. Integer variables can be assumed, in the analysis, to be real in order to solve Constraint Logic Programming (CLP) problems. 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 form $x \leq y \leq c$ as a variable $x$ is a numeric constant. It is used to check for indexes in array access.

The points $\bullet$ discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The complete proof for all theorems can be found in the author’s Master thesis [2].

The Octagon domain $\pm x + \pm y < c$ is a former numerical abstract domain based on Difference-Bound Matrices and allows us to represent invariants of the form $y = (x - a) \times c$. It simulates division by zero, out-of-bound array access or deadlock, and this allows discovering automatically common errors, such as index out of bound error when accessing the array $\text{tab}[i]$.Assertions in curly brackets $\{\cdot\}$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The hits are done as a variable $\text{tab}[i]$ is a numeric constant. It is used to check for indexes in array access.

Our method works well for reals and rationals. Integer variables can be assumed, in the analysis, to be real in order to solve Constraint Logic Programming (CLP) problems. 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 form $x \leq y \leq c$ as a variable $x$ is a numeric constant. It is used to check for indexes in array access.

The points $\bullet$ discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The complete proof for all theorems can be found in the author’s Master thesis [2].

The Octagon domain $\pm x + \pm y < c$ is a former numerical abstract domain based on Difference-Bound Matrices and allows us to represent invariants of the form $y = (x - a) \times c$. It simulates division by zero, out-of-bound array access or deadlock, and this allows discovering automatically common errors, such as index out of bound error when accessing the array $\text{tab}[i]$.Assertions in curly brackets $\{\cdot\}$ are discovered automatically by a simple static analysis and stores the hits in the array $\text{tab}[i]$. The hits are done as a variable $\text{tab}[i]$ is a numeric constant. It is used to check for indexes in array access.
Overview

20 is significantly faster than existing analyzers based on symbolic analysis, which often take hours to verify simple fully connected networks, and AI\textsuperscript{2} can handle deep convolutional networks, which are beyond the reach of existing methods.

Reading

“Based on overapproximation, AI\textsuperscript{2} can automatically prove safety properties (e.g., robustness) of realistic neural networks (e.g., convolutional neural networks).”

“Our results demonstrate that:

i. AI\textsuperscript{2} is precise enough to prove useful specifications (e.g., robustness),

ii. AI\textsuperscript{2} can be used to certify the effectiveness of state-of-the-art defenses for neural networks,

iii. AI\textsuperscript{2} is significantly faster than existing analyzers based on symbolic analysis, which often take hours to verify simple fully connected networks, and

iv. AI\textsuperscript{2} can handle deep convolutional networks, which are beyond the reach of existing methods.”

Recipe

AI\textsuperscript{2}: Safety and Robustness Certification of Neural Networks with Abstract Interpretation

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri

Department of Computer Science
ETH Zurich, Switzerland

Abstract—We present AI\textsuperscript{2}, the first sound and scalable analyzer for deep neural networks. Based on overapproximation, AI\textsuperscript{2} can automatically prove safety properties (e.g., robustness) of realistic neural networks (e.g., convolutional neural networks). The key insight behind AI\textsuperscript{2} is to phrase reasoning about safety and robustness of neural networks in terms of classic abstract interpretation. For large classifiers while maintaining a precision that suffices to prove useful properties. The analyzer must consider all possible outputs of the network over a prohibitively large set of inputs, processed by a vast number of intermediate neurons. This challenge has become critical in light of recent research [40] showing that attacks can be executed physically [41].

Recent years have shown a wide adoption of deep neural networks in safety-critical applications, including self-driving cars [12], malware detection [44], and aircraft collision avoidance detection [21]. This adoption can be attributed to the near-human accuracy obtained by these models [22, 23].

Despite their success, a fundamental challenge remains: to ensure that machine learning systems, and deep neural networks in particular, behave as intended. This challenge has become critical in light of recent research [4] showing that even highly accurate neural networks are vulnerable to adversarial examples. Adversarial examples are typically obtained by slightly perturbing an input that is correctly classified by the network, such that the network misclassifies the perturbed input. Various kinds of perturbations have been shown to successfully generate adversarial examples (e.g., [5, 13, 14, 15, 13, 10, 19, 19, 132, 14, 18]).

I. INTRODUCTION

AI\textsuperscript{2} is significantly faster than existing analyzers based on symbolic analysis, which often take hours to verify simple fully connected networks, and AI\textsuperscript{2} can handle deep convolutional networks, which are beyond the reach of existing methods.

Adversarial examples can be especially problematic when safety-critical systems rely on neural networks. For instance, it has been shown that attacks can be executed physically (e.g., [9, 14]) and against neural networks accessible only as a black box (e.g., [32, 48, 49]). To mitigate these issues, recent research has focused on reasoning about neural network robustness, and in particular on local robustness: local robustness (or robustness, for short) requires that all samples in the neighborhood of a given input are classified with the same label [11].

Many works have focused on designing defenses that increase robustness by using modified procedures for training the network (e.g., [12, 13, 27, 31, 42]). Others have developed approaches that can show non-robustness by underapproximating neural network behaviors [3] or methods that decide robustness of small fully connected neural networks [21]. However, no existing neural analyzer handles abstract interpretation, one of the most popular approaches.

Key Challenges: Scalability and Precision. The main challenge facing neural analysis of neural networks is scaling to large classifiers while maintaining a precision that suffices to prove useful properties. The analyzer must consider all possible outputs of the network over a prohibitively large set of inputs, processed by a vast number of intermediate neurons. For instance, consider the image of the digit (in Fig. 1, the network has become critical in light of recent research [40] showing that attacks can be executed physically [41].

Recent years have shown a wide adoption of deep neural networks in safety-critical applications, including self-driving cars [12], malware detection [44], and aircraft collision avoidance detection [21]. This adoption can be attributed to the near-human accuracy obtained by these models [22, 23].

Despite their success, a fundamental challenge remains: to ensure that machine learning systems, and deep neural networks in particular, behave as intended. This challenge has become critical in light of recent research [40] showing that attacks can be executed physically [41].

Recent years have shown a wide adoption of deep neural networks in safety-critical applications, including self-driving cars [12], malware detection [44], and aircraft collision avoidance detection [21]. This adoption can be attributed to the near-human accuracy obtained by these models [22, 23].

Despite their success, a fundamental challenge remains: to ensure that machine learning systems, and deep neural networks in particular, behave as intended. This challenge has become critical in light of recent research [40] showing that attacks can be executed physically [41].
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 including recursive procedures, dynamic memory allocation, and global variables. The design of Verasco, which is connected to the CompCert formally-verified C compiler so that not only the semantics of the analyzed programs is guaranteed with mathematical certitude, but also the fact that these guarantees carry over to the compiled code. Verasco establishes the absence of run-time errors such as out-of-bound array accesses, null pointer dereferences, and arithmetic exceptions. These basic properties are essential both for safety and security. Among the various verification techniques, state analysis is perhaps the one that scales best to large existing code bases, with minimal intervention from the programmers.

Static analyzers can be used in two different ways: as sophisticated bug finders, discovering potential programming errors that are hard to locate 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 performed on every line of code; this is prohibitively expensive for large programs. For verification, the analysis must be performed on only the program flows of interest; the analysis must therefore be provided. Owing to the complexity of static analyzers and of their input data (programs written in "big" programming languages), the tools used to produce these flows are themselves often verified programs that must also be given a guarantee of soundness.

Verification tools are based on a variety of techniques such as static analysis; abstract interpretation; soundness proofs; proof assistants. Verification tools are increasingly used during the development and validation of critical software. These tools provide guarantees that are stronger than those of other verification methods such as testing and code review; often stronger; and sometimes cheaper (rigorous testing can be very expensive).

Verification of critical software requires formal methods such as testing and code review, often stronger, and sometimes cheaper (rigorous testing can be very expensive). Verasco is based on a variety of techniques in static analysis, including checking, deductive program proof, and combinations thereof. The guarantees they provide range from basic semantic safety to full functional correctness. In this paper, we focus on the design and the soundness proof of Verasco, which is based on the CompCert C formally-verified compiler. We apply deductive formal verification of a static analyzer to the implementation of an abstract interpreter, using the Coq proof assistant to the implementation of a static analyzer to the implementation of an abstract interpreter, using the Coq proof assistant, to the implementation of a static analyzer. We then present the absence of run-time errors such as out-of-bound array accesses, null pointer dereferences, and arithmetic exceptions. These basic properties are essential both for safety and security.

The verification is performed in Coq, which allows us to verify the correctness of the abstract interpreter and the static analyzer using Coq’s proof assistant. The Coq proof assistant is a widely accepted framework for formal proof development. The abstract interpreter is verified to be correct with respect to the dynamic semantics of the analyzed language.

1. Introduction

To use a static analyzer as a verification tool, and obtain certification credit in regulations such as DO-178 (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), the tools used to produce these flows are themselves often verified programs that must also be given a guarantee of soundness.

One analyzer, called Verasco, is based on abstract interpretation. Verasco is an end-to-end tool that, unlike other tools such as testing and code review, provides guarantees that are stronger than those of other verification methods such as testing and code review. For program verification, in contrast, evidence of the absence of run-time errors is mandatory. The absence of run-time errors in the analyzed programs is essential both for safety and security. Among the various verification techniques, state analysis is perhaps the one that scales best to large existing code bases, with minimal intervention from the programmers.

Verification tools are based on a variety of techniques such as static analysis; abstract interpretation; soundness proofs; proof assistants. Verification tools are increasingly used during the development and validation of critical software. These tools provide guarantees that are stronger than those of other verification methods such as testing and code review; often stronger; and sometimes cheaper (rigorous testing can be very expensive).

Verification of critical software requires formal methods such as testing and code review, often stronger, and sometimes cheaper (rigorous testing can be very expensive). Verasco is based on a variety of techniques in static analysis, including checking, deductive program proof, and combinations thereof. The guarantees they provide range from basic semantic safety to full functional correctness. In this paper, we focus on the design and the soundness proof of Verasco, which is based on the CompCert C formally-verified compiler. We apply deductive formal verification of a static analyzer to the implementation of an abstract interpreter, using the Coq proof assistant to the implementation of a static analyzer. We then present the absence of run-time errors such as out-of-bound array accesses, null pointer dereferences, and arithmetic exceptions. These basic properties are essential both for safety and security.

The verification is performed in Coq, which allows us to verify the correctness of the abstract interpreter and the static analyzer using Coq’s proof assistant. The Coq proof assistant is a widely accepted framework for formal proof development. The abstract interpreter is verified to be correct with respect to the dynamic semantics of the analyzed language.

1. Introduction

To use a static analyzer as a verification tool, and obtain certification credit in regulations such as DO-178 (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), the tools used to produce these flows are themselves often verified programs that must also be given a guarantee of soundness. One analyzer, called Verasco, is based on abstract interpretation. Verasco is an end-to-end tool that, unlike other tools such as testing and code review, provides guarantees that are stronger than those of other verification methods such as testing and code review. For program verification, in contrast, evidence of the absence of run-time errors is mandatory.

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