NEURAL ABSTRACT INTERPRETATION: LEVERAGING NEURAL NETWORKS FOR AUTOMATED, EFFICIENT AND DIFFERENTIABLE ABSTRACT INTERPRETATION

BY

SHAURYA GOMBER

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Advisor:

Assistant Professor Gagandeep Singh

ABSTRACT

Abstract Interpretation is a popular technique for formally analyzing the properties of programs, neural networks, and complex real-world systems. However, designing efficient abstract transformers for expressive relational domains such as Octagon and Polyhedra is hard as one needs to carefully balance the fundamental tradeoff between the cost, soundness, and precision of the transformer for downstream tasks. Further, scalable implementations involve intricate performance optimizations like Octagon and Polyhedra decomposition. This motivates the need for the automatic generation of efficient, sound, and precise abstract transformers. Given the inherent complexity of abstract transformers and the proven capability of neural networks to effectively approximate complex functions, this thesis envisions and proposes the concept of *Neural Abstract Transformers*: neural networks that serve as abstract transformers. The Neural Abstract Interpretation (`NeurAbs`) framework introduced in this thesis provides supervised and unsupervised methods to learn efficient neural transformers *automatically*, which reduces development costs. These neural transformers can then act as a *fast* and sometimes even more *precise* replacement for slow and imprecise hand-crafted transformers. Additionally, these neural transformers are *differentiable* as opposed to the hand-crafted ones. This enables *differentiable abstract interpretation* and allows for the use of gradient-guided learning methods to solve problems that can be posed as learning tasks. We instantiate the `NeurAbs` framework for two widely used numerical domains: Interval and Octagon. Evaluations on these domains demonstrate the effectiveness of the `NeurAbs` framework to learn sound and precise neural transformers. We further demonstrate the advantages of differentiability of neural transformers by formulating the task of invariant generation as a learning problem and then using the learned neural transformers in the Octagon domain to generate valid octagonal invariants.

"To my parents, grandparents, and sister: For their infinite love and support throughout."

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Gagandeep Singh. This work would not have been possible without his constant guidance. When I began my MS, I had little knowledge of program analysis or abstract interpretation. It was through interactions with him and his courses that my interest in program analysis was sparked, and for this, I will always be thankful. Our discussions have always been a joy, and I am continually impressed by his depth of knowledge and innovative ideas. I cannot be more excited to start my PhD and continue working with him.

In addition to this thesis, during my MS, I had the opportunity to work on several other projects. I want to extend my gratitude to Professor Mahesh Viswanathan and Professor Madhusudan Parthasarathy for their guidance on those projects and to Adithya Murali for being an excellent collaborator and guide! I also thank Professor Elsa Gunter for the delightful two semesters I spent TAing her course (CS421) and Kevin Cheang, my unwaveringly supportive manager during my 2023 summer internship at AWS.

My time in Champaign would not have been nearly as enjoyable without the amazing friends I made here. First and foremost, I would like to thank Nirav for being the best flatmate I could ever hope for and for all the time we spent randomly discussing life and eating out. I am also very grateful and thankful to: Mayank, Atharv, and Abhilash for the squash/gym sessions as well as all the parties, dinners, and infinite chatter on random stuff; Pratik for his homemade food and his mature advices; Ajay for the candid and deep conversations; Ishika for her lively presence and banter at FEs every Friday; Aditya and Shradha for being awesome 501 E party hosts and Jayant, Seemandhar, and Brinda for the trips and random hangouts. I also want to thank my lab friends: Aval, Debangshu, Yasmin, and Shubham, who made working so much fun.

Lastly, and most importantly, I am immensely grateful to my parents, Dr. Meenakshi Gomber and Dr. Gaurav Gomber, as well as my grandparents, Saroj Gomber and DV Gomber, and my sister, Snigdha Gomber. These five individuals are my pillars of strength, showering me infinite love and support throughout my life. Whatever I achieve or have achieved is solely due to their unwavering support and guidance. I will always be indebted to them and aspire to make them proud.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Abstract Interpretation [1] is a popular technique for formally analyzing the properties of programs [2, 3], neural networks [4, 5], and complex real-world systems [6]. Abstract Interpretation works by soundly approximating the *concrete* semantics of the system (concrete domain $\mathcal{C}$) within a "suitably finite" domain, called the *abstract domain* ($\mathcal{A}$). The "finiteness" of the abstract domain allows us to reason about all possible executions of the systems efficiently. For example, consider a case where we map program variables, which can assume integer values, to elements in the abstract domain $\mathcal{A}_{Even/Odd} = \{Even, Odd, \top, \bot\}$. Variables that are definitively even can be represented as $Even$, while those that are definitively odd can be denoted as Odd. The symbol $\top$ is used to denote variables that might be either even or odd, and $\bot$ represents the state where we have no information about the variables (such as uninitialized variables). Such a domain can help us reason about properties that depend on the odd/even parity of program variables.

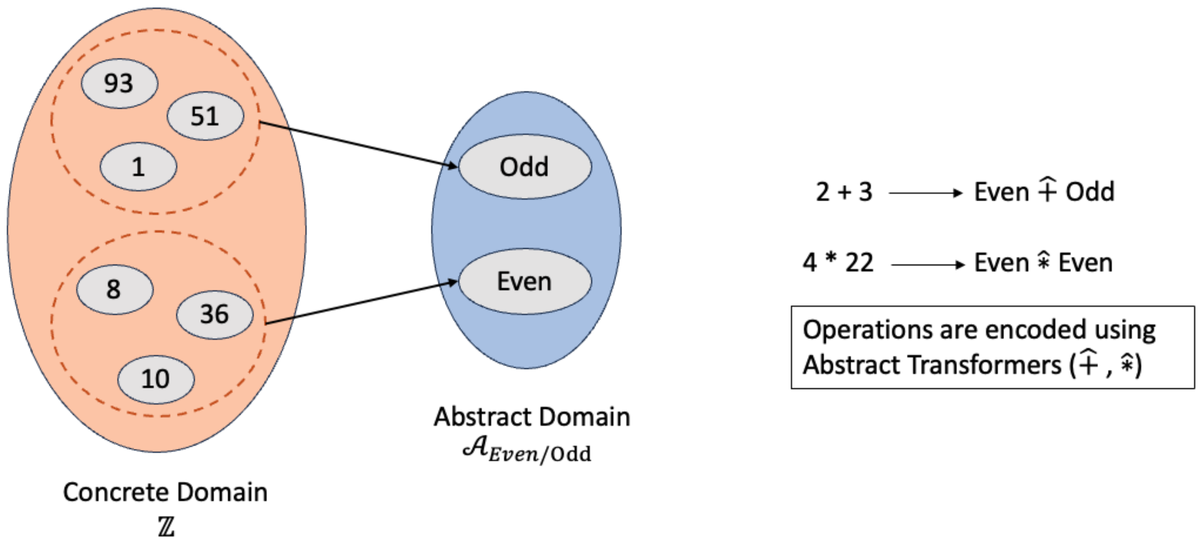

93
51
1
8
36
10
Concrete Domain
ℤ
Odd
Even
Abstract Domain
$\mathcal{A}_{Even/Odd}$
2 + 3 → Even $\hat{+}$ Odd
4 * 22 → Even $\hat{*}$ Even
Operations are encoded using Abstract Transformers ($\hat{+}$, $\hat{*}$)

Figure 1.1: Interpreting the set of integers in the Even/Odd domain $\mathcal{A}_{Even/Odd}$

Analyzing programs in the abstract domain also requires functions $\hat{op}$ that transform one abstract state to another, corresponding to operations op in the concrete domain. $\hat{op}$ is known as the *Abstract Transformer* corresponding to op and should soundly approximate the behavior of op for correctness of the analysis. In our previous example of $\mathcal{A}_{Even/Odd}$, say there is a program statement $z = x + y$, where $+$ is the addition operator on the integers. As we abstract the variables x, y into elements from $\mathcal{A}_{Even/Odd}$, we need to define $\hat{+}$ that works on elements $\hat{x}$, $\hat{y}$ from the abstract domain and capture the semantics of $+$. For example, as we know, adding two even numbers gives us an even number. So, $\mathit{Even} \mathbin{\hat{+}} \mathit{Even} = \mathit{Even}$. Similarly, it is easy to see that the following definition of $\hat{+}$ captures the concrete semantics of $+$ soundly:

$\hat{x}$ \ $\hat{y}$	*Even*	*Odd*	$\top$	$\bot$
Even	*Even*	*Odd*	$\top$	$\bot$
Odd	*Odd*	*Even*	$\top$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table 1.1: Semantics of $\hat{+}$ for $\mathcal{A}_{Even/Odd}$

The choice of the abstract domain is usually based on the specific properties to be proven. For example, in the analysis of numerical programs, abstract domains such as Octagons [7] and Polyhedra [8] are beneficial for verifying intricate program properties because they account for inter-variable dependencies, unlike the Interval domain [9], which solely represents variable bounds. However, designing efficient abstract transformers for these expressive relational domains is not trivial and is tedious because of the following reasons:

1. **Ensuring Soundness.** While designing abstract transformers, it is necessary to ensure its soundness on all elements in the abstract domain $\mathcal{A}$. This is essential to maintain the soundness of the abstract interpretation analysis.

2. **Computational complexity.** Sound and most precise implementations of abstract transformers for operations such as join in the Polyhedra and the Octagon domain are computationally expensive. Polyhedra join involves taking the convex hull of two polyhedra, whose time complexity is exponential in the number of variables in the program ($O(nm^{2^{n+1}})$ where n is the number of variables and m is the number of constraints) [10]. Octagon join involves taking the closure of the octagon whose time complexity is $O(n^3)$ ([7]). Using these implementations makes the task of program analysis expensive and, thus, less scalable.

3. **Intricate optimizations.** Scalable implementations of transformers require performance optimizations like Octagon and Polyhedra decomposition [10, 11]. Implementing such optimizations can be a very intricate task, thus requiring a lot of manual effort and increasing the chances of soundness bugs.

4. **Handling Imprecision.** For some operators, like affine assignment in the octagon domain ([7]), it is hard to implement efficient abstract transformers without losing pre-

cision. The efficient implementations of such transformers can thus be very imprecise, which makes the overall task of program analysis using these transformers less precise.

To mitigate these issues and to automate the task of designing efficient abstract transformers, we propose a data-driven learning approach to generate efficient abstract transformers. Given the inherent complexity of abstract transformers and the proven capability of neural networks to effectively approximate complex functions [12, 13], this thesis introduces the Neural Abstract Interpretation (`NeurAbs`) framework to learn **Neural Abstract Transformers**: neural networks that serve as the abstract transformers. The `NeurAbs` framework has the following advantages:

1. **Automatic Generation of Transformers.** The `NeurAbs` framework proposes supervised and unsupervised approaches to learn neural abstract transformers from data while also giving ways to fine-tune and balance the soundness-precision tradeoff. This provides an automated way to generate transformers and eases development costs.

2. **Fast and Precise Transformers.** The learned neural transformers can act as a fast and sometimes even more precise replacement for slow and imprecise hand-crafted transformers, thus making the downstream analysis tasks more scalable and precise. Unsound cases can be handled by resorting to hand-crafted transformers' outputs.

3. **Differentiable Transformers.** Additionally, the neural transformers are *differentiable* as opposed to the hand-crafted ones, enabling their use with gradient-guided learning methods, which can be beneficial for tasks that can be posed as learning problems. This opens up the avenue for *differentiable abstract interpretation.* One example of this can be the task of generating valid octagonal invariants for a loop program, which we describe in Section 5.3.

The main contributions of this thesis are summarized as follows.

1. **Problem Formulation.** We pose the problem of learning sound and precise abstract transformers as an optimization problem (Sec. 3.1). We also point out why the general optimization problem is hard to solve.

2. **General Framework With Relaxations.** We introduce a general framework `NeurAbs` (Sec. 3) that proposes supervised (Sec. 3.3) and unsupervised (Sec. 3.4) learning approaches as a relaxation of the general optimization problem. To the best of our knowledge, *we are the first work to propose such relaxation*, thus enabling the learning and use of neural abstract transformers.

3. **Evaluation.** We instantiate our `NeurAbs` framework for the Interval and the Octagon domain (Sec. 4). We demonstrate the effectiveness of our supervised and unsupervised learning methods by learning sound and precise neural transformers for operators in the Interval (Sec. 5.1) and Octagon domains (Sec. 5.2). We also demonstrate how the differentiability of the neural transformers can help us pose and solve the task of generating valid octagonal invariants for loop programs as a learning problem (Sec. 5.3).

The thesis is organized as follows:

- Chapter 2 provides essential background knowledge on Abstract Interpretation and Abstract Transformers.
- Chapter 3 explains the supervised and unsupervised learning relaxations proposed by our `NeurAbs` framework.
- In Chapter 4, we demonstrate how our `NeurAbs` framework can be instantiated for the Interval and Octagon domains.
- In Chapter 5, we evaluate the efficacy of our framework in learning sound and precise neural transformers for the Interval and Octagon domains. We also demonstrate the benefits of the differentiability of the learned neural transformers by posing the task of invariant generation as a learning problem.
- Chapter 6 covers some related works in the area of learning abstract transformers and neural surrogates.
- Chapter 7 serves as the conclusion of the thesis, where we discuss potential improvements and future avenues for exploration.

CHAPTER 2: BACKGROUND

This section begins with a brief introduction to Abstract Interpretation, highlighting the concepts of *Abstraction and Concretization* functions and their required relationship through the *Galois connection* to maintain the *soundness* of abstract interpretation. Following this, we introduce Abstract Transformers, discussing their *soundness and precision*—key aspects needed to understand the problem we address. We then discuss how Abstract Interpretation allows for sound but incomplete analysis of programs. The section wraps up with an overview of two abstract numerical domains utilized in our evaluation: Interval and Octagon.

2.1 ABSTRACT INTERPRETATION

Proving various properties of programs and establishing their correctness is undecidable in general. So, to make the verification process tractable, program analyzers typically work on an *abstraction* of the program, which over-approximates the semantics of the original program. This technique is known as Abstract Interpretation [1]. Abstract Interpretation is the theory of the sound approximation of the semantics and states of programs (C*oncrete Domain* $\mathcal{C}$) through elements belonging to an alternative domain, commonly referred to as the *Abstract Domain* ($\mathcal{A}$). The core concept of Abstract Interpretation is that it effectively "partially executes" the program within the abstract domain $\mathcal{A}$. The abstract domain $\mathcal{A}$ is chosen in a way such that it is "suitably finite". This "finiteness" ensures that analyzing the program's semantics and states within the domain $\mathcal{A}$ provides a concise yet sound analysis of all potential program executions. This enables us to provide formal guarantees concerning the presence or absence of certain bugs and the verification of specific properties.

Abstraction and Concretization Functions. The *abstraction function* $\alpha : \mathcal{P}(\mathcal{C}) \rightarrow \mathcal{A}$ maps sets of elements in the concrete domain to values in the abstract domain. On the other hand, the *concretization function* $\gamma : \mathcal{A} \rightarrow \mathcal{P}(\mathcal{C})$ maps abstract elements back to the set of concrete elements they represent. For instance, the abstraction and the concretization functions used for abstracting integers $\mathbb{Z}$ using the *interval abstract domain* $\mathcal{A}_{Intv}$ are illustrated in Fig 2.1. In this case, α maps sets of integers to *the smallest interval* that contains all integers from the set. For e.g., $\alpha(\{-2, 1, 2\}) = [-2, 2]$. Conversely, γ maps the intervals to the *largest set of integers* that the interval abstracts. So, $\gamma([-2, 2]) = \{-2, -1, 0, 1, 2\}$. This also demonstrates the loss of precision that arises when using abstractions as the set $\{-2, 1, 2\}$ was abstracted using $[-2, 2]$, which, when concretized, gives $\{-2, -1, 0, 1, 2\}$. This set has all integers from the original set, but also new integers that are added because of the

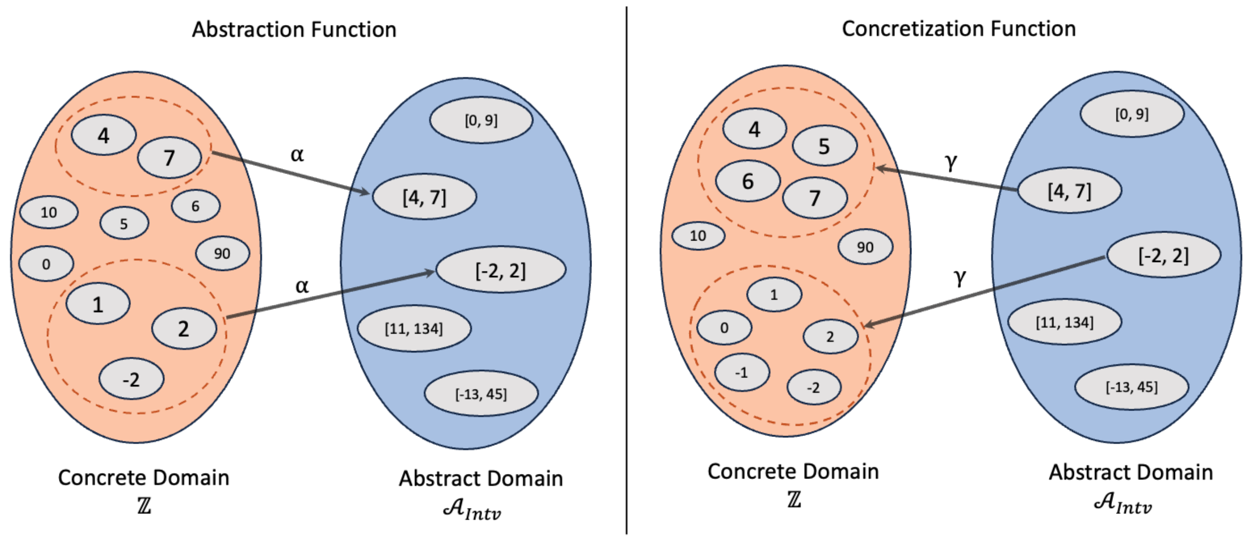


Figure 2.1: Abstraction and Concretization Functions to abstract integers in the interval domain. It also demonstrates the loss of precision introduced by concretizing back the abstract elements as the set returned after concretizing $[-2, 2]$ has elements that were not there in the original set.

loss of precision introduced when concretizing the abstract elements.

Galois Connection. We require that our analysis using abstract interpretation is *sound*, i.e., the analysis in the abstract domain *safely over-approximates* the semantics of the concrete domain. This can be ensured if the concrete and abstract domains are connected by the Galois Connection, which is defined as follows:

Definition 2.1. Let $\mathbb{P}_{\mathbb{C}} = (\mathcal{P}(\mathcal{C}), \sqsubseteq_C)$ be the poset on the power set of states in the concrete domain $\mathcal{C}$ and $\mathbb{P}_{\mathbb{A}} = (\mathcal{A}, \sqsubseteq_A)$ be the poset on the set of states in the abstract domain $\mathcal{A}$, then α and γ are connected by the *Galois connection* iff:

$$\forall x \in \mathcal{P}(\mathcal{C}).\ \forall z \in \mathcal{A}.\ \alpha(x) \sqsubseteq_A z \iff x \sqsubseteq_C \gamma(z) \tag{2.1}$$

Intuitively, this means that α and γ respect the orderings of $\mathcal{P}(\mathcal{C})$ and $\mathcal{A}$ as illustrated in Fig 2.2. The following directly follows from the above definition (by substituting $z = \alpha(x)$):

$$\forall x \in \mathcal{P}(\mathcal{C}).\ x \sqsubseteq_C \gamma(\alpha(x)) \tag{2.2}$$

This means that, for the soundness of the analysis, the set of concrete states obtained by concretizing the abstraction of any set should at least contain that set. The rest of the states not there in the original set lead to the imprecision discussed above.

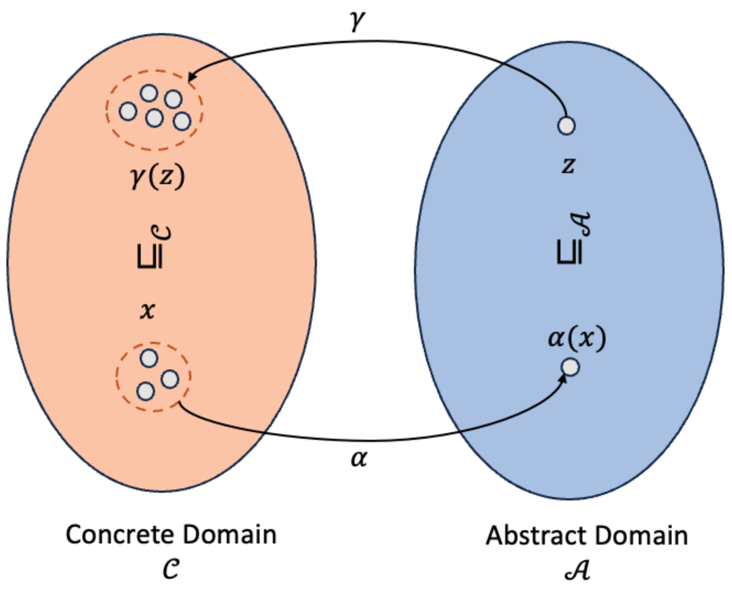


Figure 2.2: Galois Connection

2.2 ABSTRACT TRANSFORMERS

Analyzing programs in the abstract domain requires functions that transform elements in the abstract domain as a result of operations applied in the concrete domain. Such functions are known as Abstract Transformers. Abstract Transformer corresponding to an operation op in the concrete domain ($\mathcal{C}$) is a function $\hat{op} : \mathcal{A} \to \mathcal{A}$ that captures the effect of applying op to concrete states corresponding to an abstract state in $\mathcal{A}$.

Soundness of Abstract Transformers. For the analysis to be sound, the abstract transformer $\hat{op}$ should be sound, i.e., it should over-approximate the output of the concrete operator op. When the powerset of concrete states $\mathcal{P}(\mathcal{C})$ is related to $\mathcal{A}$ by a Galois connection $\mathcal{P}(\mathcal{C}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$, the soundness condition for $\hat{op}$ can be mathematically defined as:

$$\forall z \in \mathcal{A}.\ \alpha(op(\gamma(z))) \sqsubseteq_A \hat{op}(z) \tag{2.3}$$

This means that if we start from any abstract state z and perform the following:

1. Concretize it to get the set of concrete states represented by it: $\gamma(z)$.
2. Get the concrete states obtained by applying op on those concrete states: $op(\gamma(z))$.
3. Get the abstraction for the set of concrete states obtained in Step 2: $\alpha(op(\gamma(z)))$.

Then the value returned by the abstract transformer $\hat{op}$ should always over-approximate

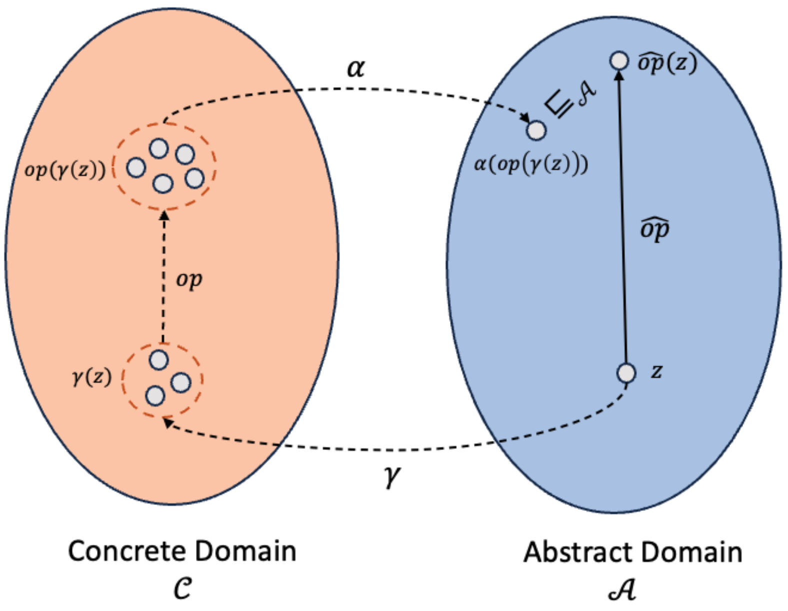


Figure 2.3: Soundness of Abstract Transformer $\hat{op}$

$\alpha(op(\gamma(z)))$, because intuitively, $\alpha(op(\gamma(z)))$ represents the *smallest* abstract element that covers all possible concrete values that can be generated by *op*. Cousot et. al. in [1] refer to $\alpha(op(\gamma(z)))$ as the *most-precise abstract transformer* $\hat{op}^{\#}$ (or the "best transformer for *op*"). So, any abstract transformer for operator *op* is sound if it over-approximates $\hat{op}^{\#}$.

Given that α and γ are related by the Galois connection (Eq 2.1), the soundness condition in Eq. 2.3 can be re-written only in terms of γ as follows:

$$\forall z \in \mathcal{A}.\ op(\gamma(z)) \sqsubseteq_C \gamma(\hat{op}(z)) \tag{2.4}$$

We will be using this definition of the soundness of abstract transformers in this work.

Precision of Abstract Transformers. Soundness of abstract transformers is a necessary condition for sound analysis using abstract interpretation. However, sound abstract transformers can be naively defined by always returning $\top$ (top) as the output of the abstract transformer. The top element of a set (lattice) is the *greatest* element in the set. Intuitively, this means that the abstract transformer always returns the abstract state that corresponds to all possible concrete states (the complete set $\mathcal{C}$). Clearly, such transformers will always maintain soundness. However, the significant imprecision that results makes it less useful for subsequent analysis tasks. For practical applications, it is necessary for the transformer to be both sound and as precise as possible. The *precision* of $\hat{op}$ is essentially indicative of the degree of over-approximation due to $\hat{op}$ and can be quantified by some measure of the size of the abstract element computed by $\hat{op}$.

2.3 SOUND, BUT INCOMPLETE ANALYSIS

Using an example, let's demonstrate how abstract interpretation facilitates the sound (but incomplete) analysis of programs. Consider, for example, the simple function f in Fig 2.4. Say we need to verify that the output c of the function always satisfies $c \geq -2$ for all possible input pairs (x, y), where x varies between $[0, 7]$ and y from $[3, \infty]$. Due to the infinite possibilities for input pairs, manually testing f for each combination to validate this property is impractical. This is where Abstract Interpretation is utilized. Employing the Interval Abstract Domain, , we can represent all possible values of x and y with the abstract elements $\hat{x} = [0, 7]$ and $\hat{y} = [3, \infty]$. This enables an *abstract execution* of the function within the Interval domain, analyzing it line by line as follows:

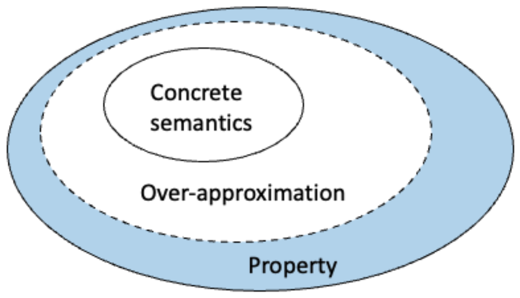
```
def f(x,y):
    a = x + y;
    b = x - y;
    c = a - b;
    return c
```

Figure 2.4: Simple program

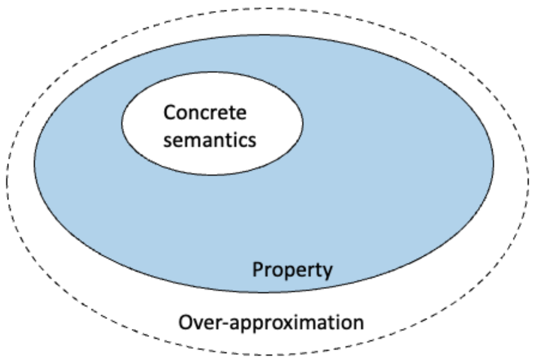
$$\hat{a} = \hat{x} \mathbin{\hat{+}} \hat{y} = [0,7] \mathbin{\hat{+}} [3,\infty] = [3,\infty] \tag{2.5}$$
$$\hat{b} = \hat{x} \mathbin{\hat{-}} \hat{y} = [0,7] \mathbin{\hat{-}} [3,\infty] = [0,7] \mathbin{\hat{+}} [-\infty,-3] = [-\infty,4] \tag{2.6}$$
$$\hat{c} = \hat{a} \mathbin{\hat{-}} \hat{b} = [3,\infty] \mathbin{\hat{-}} [-\infty,4] = [3,\infty] \mathbin{\hat{+}} [-4,\infty] = [-1,\infty] \tag{2.7}$$

Here, $\hat{+}$ and $\hat{-}$ denote the sound abstract transformers for the $+$ and $-$ operators, respectively. The analysis results indicate that the value of c spans from $[-1, \infty]$. Given that both the abstract domain and the sound abstract transformers over-approximate the actual values, we can conclude that all potential values for c are indeed within the range of $[-1, \infty]$. This over-approximation confirms that c will always be greater than or equal to -2, hence proving the property.

Incompleteness. The analysis based on abstract interpretation is sound; therefore, if a property is verified using this method, the property is indeed valid. However, the analysis is not complete, i.e., failure to verify a property does not necessarily imply that the property is invalid. For example, in the program above, it can be checked that $c = 2 * x$, and so for x in $[0, 7]$, c would be from $[0, 14]$. So properties like $c \geq 0$ and $c \leq 14$ also hold. But our analysis using the interval abstract domain gives us the range of c as $[-1, \infty]$, which can not be used to prove $c \geq 0$ or $c \leq 14$, even if they hold. However, note that the analysis did a correct over-approximation as $([0, 14] \in [-1, \infty])$ but the imprecision introduced by the



(a) If over-approximation proves the property, the property holds.



(b) The property can also hold even if the over-approximation does not prove it.

Figure 2.5: Analysis using over-approximation is sound but not complete

over-approximation does not allow us to verify some properties which actually hold.

2.4 NUMERICAL ABSTRACT DOMAINS

Numerical abstract domains abstract a set of numbers. While analyzing programs, these sets of numbers can be the possible values that the program variables can take. Numerical abstract domains can be used to prove various properties of numerical programs. Next, we introduce some commonly used numerical domains.

2.4.1 Interval Domain

In the Interval domain, a set of numbers is abstracted by the smallest interval that contains those numbers. For e.g, the set $\{-1.2, 2.3, 4.9, 2\}$ will be abstracted by the interval $[-1.2, 4.9]$. If a program has n variables, then we would have n intervals where the i^{th} interval would abstract the set of possible values of the i^{th} variable. The interval domain, thus, is a non-relational domain, as the relationship between the variables is not maintained due to the independent representation as intervals. Consider a simple program with two statements: $x = a + b$; $y = a - b$. If the initial interval for a and b are $[1, 2]$, then the resulting interval for x will be $[2, 4]$ and for y, it will be $[-1, 1]$ (given by $[1, 2] + [-2, -1]$). The final state $x \in [2, 4]$ and $y \in [-1, 1]$ consists of state where $x = 4$ and $y = 1$. But note that this is impossible in the program as if $x = 4$, a, and b have to be 2, and thus y has to be 0. However, the Interval domain does not maintain the relationship between the variables and treats them independently. This is also why it is very imprecise (as seen in Fig 2.6).

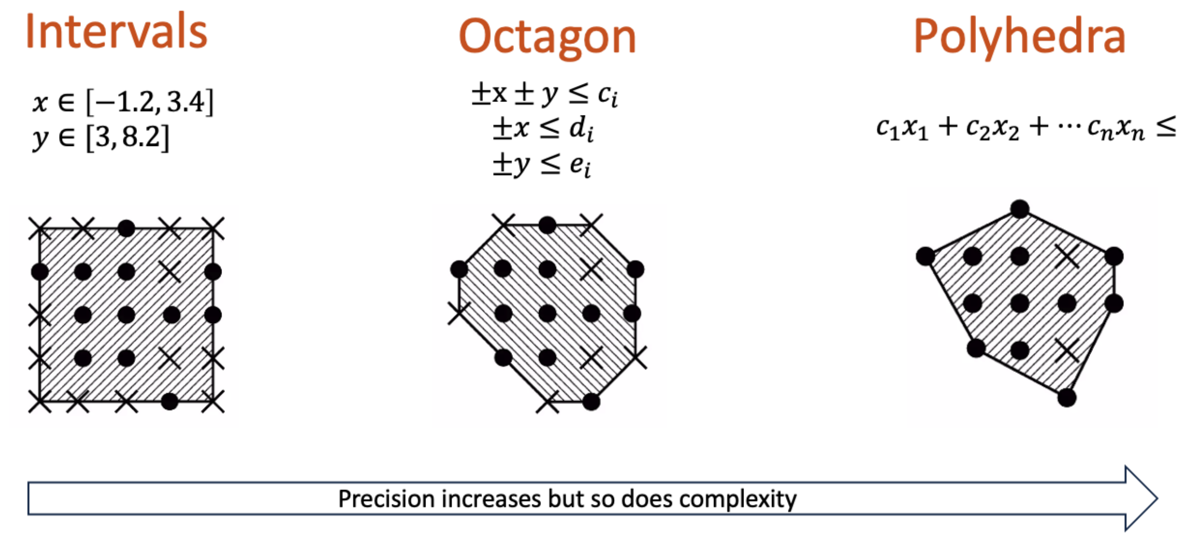


Figure 2.6: Abstracting the set of black dots using the different abstract domains ([14]). The crosses indicate the extra points that are added in the abstraction and that lead to imprecision. As we go right, the precision increases but so does the domain complexity.

Abstract transformers for the interval domain operate on intervals. For e.g., if the program has a statement $z = abs(x)$, then the abstract transformer for abs (given by $\hat{abs}$) would take in an interval $[l_1, u_1]$ and return a new interval $[l_2, u_2]$ such that it contains absolute of all values in $[l_1, u_1]$. For e.g, $\hat{abs}([1, 3])$ would return $[1, 3]$, $\hat{abs}([-4, -1])$ would return $[1, 4]$ and that $\hat{abs}([-10, 2])$ would return $[0, 10]$. It is easy to check that for a general $[l, u]$, $\hat{abs}([l, u])$ is given by $[max(max(0, l), -u), max(-l, u)]$. Similarly, the abstract transformer for the join of two intervals $[l_1, u_1]$ and $[l_2, u_2]$ will return an interval that contains both the intervals and is given by $[min(l_1, l_2), max(u_1, u_2)]$.

2.4.2 Octagon Domain

In the octagon domain, the set of possible program states is abstracted using a *octagon shape*. Given program variables $v_1, v_2, \ldots v_n$, the octagon shape is represented by a set of inequalities between the variables where the inequalities can only be of the following types:

1. $\pm v_i \pm v_j \leq c_{ij}$: Between any 2 variables and the coefficients can only be ± 1.

2. $\pm v_i \leq d_i$: Bounds on the positive or negative value of a variable.

The Octagon domain is a weakly relational domain as it allows a limited number of relations to be captured and, thus, is more precise than the Interval domain.

Abstract transformers for the octagon domain operate on octagons. For example, the join of two octagons oct_1 and oct_2 returns an octagon that contains both the octagons.Two octagons can be joined by taking the max of all inequality constants, i.e., if oct_1 has $v_i - v_j \leq c_1$ and $v_i - v_j \leq c_2$, then the join will have $v_i - v_j \leq max(c_1, c_2)$. If the inequality is not there in either one of them, then it would not be in the join as well. However, to keep the results of the join precise, the closure operation is first performed on both the octagons. The closure operator *tightens* the inequalities in an octagon by making the explicit constraints implicit and is frequently used to make octagon operations precise. However, it is computationally expensive with time complexity $O(n^3)$ ([7]). Other transformers, like affine assignment in the octagon domain, take an octagon o and return the resultant octagon o' after computing expressions such as $z = a * x + b * y$.

2.4.3 Polygon Domain

In the polygon domain, program states are abstractly represented by a *polygon shape*, which is defined through a set of inequalities among program variables $v_1, v_2, \ldots, v_n$. This domain does not impose constraints on the types of inequalities used, unlike the octagon domain. As such, the polygon domain qualifies as a relational domain that precisely encapsulates all relationships between the variables. The lack of restrictions on the inequalities allows for high precision but also leads to the complexity of abstract transformers, such as the join operation. Specifically, performing a join in the polyhedra domain involves calculating the convex hull of two polyhedra, a process with an exponential time complexity relative to the number of variables, expressed as $O(nm^{2^{n+1}})$, where n is the number of variables and m is the number of constraints [10].

CHAPTER 3: GENERAL FRAMEWORK

In this chapter, we will first formulate the problem of learning sound and precise abstract transformers as an optimization problem and discuss why it is a hard problem to solve. We then introduce the novel concept of **Neural Abstract Transformers**, which are neural networks that serve as abstract transformers. We then propose *supervised and unsupervised relaxations* of the general optimization problem, which enables the learning and use of neural abstract transformers.

3.1 ABSTRACT TRANSFORMERS LEARNING PROBLEM

The general problem of learning abstract transformers that are simultaneously sound and precise is notably difficult. For instance, consider the task of learning the *sound* and *most-precise* abstract transformer $\hat{op}$ from a set of functions $\mathcal{F}$ for an operator *op*. As defined by Cousot et. al. in [1], we represent the "most-precise abstract transformer" for *op* as $\hat{op}^{\#}$. Cousot et. al. just provide a specification for $\hat{op}^{\#}$, and in general, there is no way to compute it. The goal then is to learn $\hat{op}$ such that its output is sound for all possible inputs in the abstract domain, and the output is as close to the *most-precise* abstract transformer $\hat{op}^{\#}$. This can be posed as the following optimization problem:

$$\hat{op} = \min_{f \in \mathcal{F}} \sum_{a \in \mathcal{A}} \mathcal{L}_P(\hat{op}^{\#}(a),\ f(a)) \ \text{ s.t.} \sum_{a \in \mathcal{A}} \mathcal{L}_S(op,\ a,\ f(a)) = 0 \tag{3.1}$$

where:

1. $\mathcal{L}_S(op, a, f(a))$ is a function that measures the soundness of f. Thus, $\mathcal{L}_S(a, f(a))$ returns 0 if $f(a)$ is a sound approximation of the effect *op* on a, i.e., $op(\gamma(a)) \sqsubseteq_C \gamma(f(a))$, where γ is the conretiziation function. $\mathcal{L}_S(a, f(a))$ returns 1 if $f(a)$ is a unsound. This means that $\sum_{a \in \mathcal{A}} \mathcal{L}_S(a,\ f(a)) = 0$ ensures that f is sound for all elements $a \in \mathcal{A}$.

2. $\mathcal{L}_P(a_1, a_2)$ is the precision measure and can be any metric to measure how "big" is a_2 compared to a_1. For instance, for the Interval domain, this can be the difference of the interval sizes of a_1 and a_2. This means that the minimizing the term $\sum_{a \in \mathcal{A}} \mathcal{L}_P(\hat{op}^{\#}(a),\ f(a))$ allows the problem to find an abstract transformer which is closest in precision to the "most-precise abstract transformer" $\hat{op}^{\#}$. Also, note that if $\hat{op}^{\#} \in \mathcal{F}$, then the solution to the above optimization problem would be $\hat{op}^{\#}$.

Inherent complexity of the above formulation. Though the above formulation to learn abstract transformers is correct, solving the optimization problem is complex due to the following reasons:

1. The set $\mathcal{A}$ of all possible abstract elements has infinite size in most cases. This makes the search for a function that satisfies the soundness and precision properties of all the abstract elements difficult.

2. Cousot et. al. (in [1]) just provide a specification for $\hat{op}^{\#}$, and do not give a way to compute it. This makes computing $\mathcal{L}_P$ non-trivial.

3. Computing $\mathcal{L}_S$ is usually done by finding counter-example e to soundness such that $op(\gamma(e)) \not\sqsubseteq_C \gamma(\hat{op}(e))$. This is done by encoding this condition, along with the semantics of op and γ in SMT [15], and then using an SMT solver. This makes computing $\mathcal{L}_S$ expensive and makes the use of gradient-guided learning methods to solve the optimization problem infeasible due to the non-differentiability of such solvers.

3.2 NEURAL ABSTRACT TRANSFORMERS

As discussed above, the optimization problem to learn sound and precise abstract transformers (given in Eq. 3.1) is hard to solve, which makes the task of automating the generation of abstract transformers difficult. To mitigate this, we relax the optimization problem described above and propose supervised and unsupervised approaches to learn a novel type of abstract transformers, which we term as **Neural Abstract Transformers**. Neural Abstract Transformers are neural networks that, when trained to enforce soundness and precision, can serve as abstract transformers. To the best of our knowledge, we are the first work to propose such relaxations of the general optimization problem, thus enabling the learning and use of neural abstract transformers.

Utilizing neural networks redefines the problem of generating sound and precise abstract transformers into a more tractable problem of learning neural networks that satisfy the soundness and precision constraints of the abstract transformers. Despite the inherent complexity of these abstract transformers, works such as [12, 13] have demonstrated the ability of neural networks to approximate complex functions effectively. This approach distinguishes our approach from works like [16, 17] that employ symbolic methods to develop abstract transformers from functions specified by a Domain Specific Language (DSL). Since sound and precise abstract transformers for operations like affine assignments in the octagon domain cannot be straightforwardly represented by simple DSL functions, the ability of neural networks to approximate complex functions can enable the efficient creation of neural transformers for such cases. Thus, using neural abstract transformers has the following benefits:

1. **Automatic Generation of Abstract Transformers.** Our supervised and unsupervised learning relaxations allow for the automated generation of transformers with varying soundness and precision, which eases development costs.

2. **Efficient Transformers.** These neural transformers can be faster alternatives for computationally expensive transformers like octagon join. For operations such as affine assignments within the octagon domain, where hand-crafted transformers often lack precision, these neural transformers can even serve as a more precise alternative.

3. **Differentiable Abstract Interpretation.** Neural transformers have the added benefit of being differentiable. This enables their use with gradient-guided learning methods, which can then be used to solve tasks like invariant generation that can be posed as learning problems.

Having outlined the benefits of Neural Abstract Transformers, we now discuss the supervised and unsupervised learning methods to train these neural transformers.

3.3 SUPERVISED LEARNING OF NEURAL TRANSFORMERS

In this section, we introduce a supervised learning approach for training neural abstract transformers. This method can train neural transformers for operators that already have hand-crafted sound and precise abstract transformers. As previously mentioned, these neural transformers can then act as a faster alternative to the hand-crafted transformers and also enable differentiable analysis.

Learning Problem. Given a dataset $\mathcal{D} = \{X_i, y_i\}$ representing input-output of an abstract transformer $\hat{op}$ (for concrete operator op) in some abstract domain $\mathcal{A}$, we pose the learning of the neural abstract transformer $\hat{op}^*$ as the following optimization problem:

$$\min_{\theta} \mathbb{E}_{(X_i, y_i)\sim D} \left[\alpha * \mathcal{L}'_S(y_i,\ \hat{op}^*(X_i;\theta)) + \beta * \mathcal{L}'_P(y_i,\ \hat{op}^*(X_i;\theta))\right] \tag{3.2}$$

which is based on the following components:

1. **Soundness Loss.** $\mathcal{L}'_S$ controls the soundness of the neural transformer. As we have the ground truth outputs y_i of the abstract transformer, soundness can be ensured by ensuring that the output of the model $\hat{op}^*(X_i;\theta)$ over-approximates y_i, i.e., $\hat{op}^*(X_i;\theta) \sqsubseteq_A y_i$. This follows from the following theorem (proved in Appendix A.1):

Theorem 3.1. If y_i is a sound output of an abstract transformer $\hat{op} : \mathcal{A} \rightarrow \mathcal{A}$ on some input x_i and $y_i \sqsubseteq_A y'_i$, then y'_i is also a sound output of $\hat{op}$ on x_i.

So, we define $\mathcal{L}'_S$ such that, for states a_1, a_2 in the abstract domain $\mathcal{A}$, $\mathcal{L}'_S(a_1, a_2) = 0$ implies that a_2 over-approximates a_1, i.e. $a_1 \sqsubseteq_A a_2$. The condition $a_1 \sqsubseteq_A a_2$ holds if a_1 is contained in a_2, i.e. the concretization of a_1 is contained in that of a_2 ($\gamma(a_1) \sqsubseteq_C \gamma(a_2)$). If $\mathcal{L}'_S(a_1, a_2) \neq 0$, then $\mathcal{L}'_S(\geq 0)$ gives a differentiable approximation of the size of the set $\gamma(a_1) \setminus \gamma(a_2)$ and minimizing $\mathcal{L}'_S(a_1, a_2)$ ensures a_2 over-approximates a_1. For instance, for the Interval domain, if the ground truth is $[0, 5]$ and the model's output is $[0, 4]$, then as $[0, 5] \not\sqsubseteq [0, 4]$, soundness loss can be given by $5 - 4 = 1$, which will guide the model to increase the upper bound (4) of the output. Minimizing $\mathcal{L}'_S(y_i,\ \hat{op}^*(X_i; \theta))$ ensures that the output of the neural transformer over-approximates the ground truth, thus ensuring the soundness of the learned abstract transformer.

Note that this is easier to compute differentiably than $\mathcal{L}_S(op, a, f(a))$ in Eq. 3.1 as that involves checking if $op(\gamma(a)) \sqsubseteq_C \gamma(f(a))$ and thus requires encoding the semantics of *op* as well. As we are in the supervised setting and already have the ground truths for the abstract transformers, we do not explicitly need the semantics of *op*.

2. **Precision Loss.** $\mathcal{L}'_P$ controls the precision of the neural transformer. Say we have a measure $\mathcal{M}(a)$ of the size of the set $\gamma(a)$ abstracted by an element $a \in \mathcal{A}$. For states a_1, a_2 in the abstract domain $\mathcal{A}$, $\mathcal{L}'_P(a_1, a_2)$ measures how big $\mathcal{M}(a_2)$ is as compared to $\mathcal{M}(a_1)$. As discussed above, we need that $\gamma(a_1) \sqsubseteq_C \gamma(a_2)$ to ensure soundness. But soundness can be ensured if the transformer always outputs $\top_{\mathcal{A}}$ (top element of lattice $\mathcal{A}$ that represents the set of all concrete states) as $\forall a \in \mathcal{A}.\ \gamma(a) \sqsubseteq_C \gamma(\top_{\mathcal{A}})$. However, this is not beneficial for downstream analysis tasks.

   So, to ensure precision, $\mathcal{L}'_P(a_1, a_2)$ (≥ 0) gives a differentiable approximation of the how big a_2 is as compared to a_1, i.e. $\max(\mathcal{M}(a_2) - \mathcal{M}(a_1),\ 0)$. For instance, for the Interval domain, the precision loss can be measured using the difference of interval sizes: $\mathcal{M}([l, u]) = \max(u - l, 0)$. Note that we take *max* with 0 as the interval is empty if $l > u$. Minimizing $\mathcal{L}'_P(y_i,\ \hat{op}^*(X_i; \theta))$ thus ensures that the output of the neural transformer is closer in size to that of the ground truth.

   Also, note that this is easier to compute differentiably than $\mathcal{L}_P(\hat{op}^{\#}(a),\ f(a))$ in Eq. 3.1 as we do not need $\hat{op}^{\#}$ to measure precision. Instead, we rely on the ground truth outputs from the abstract transformer to measure the *imprecision* between the transformer and the learned neural transformer and then use it to enforce precision.

3. **Soundness & Precision Weights.** α, β are the soundness and precision weights, respectively, and let us control the degree of soundness and precision required for the transformer. This can be adjusted based on the specific downstream analysis task for which the neural transformer will be used.

As we will demonstrate in the next sections, $\mathcal{L}'_S(a_1, a_2)$ and $\mathcal{L}'_P(a_1, a_2)$ can be computed easily and differentiably for abstract domains such as Intervals and Octagons.

Datasets. Datasets for the supervised learning method described above can be generated by executing the hand-crafted abstract transformers on random states from the abstract domain $\mathcal{A}$ or on abstract states collected from running analysis on some downstream tasks. Some abstract transformers can be computationally expensive, but the data collection task has to be done only once. The dataset can then be used to learn various neural transformers with varying soundness and precision for the same abstract transformer.

3.4 UNSUPERVISED LEARNING OF NEURAL TRANSFORMERS

In this section, we describe an unsupervised learning algorithm that can be used to train neural abstract transformers. As opposed to the supervised learning relaxation, we do not have the ground truth outputs from the abstract transformers here. Instead, we rely on the semantics of the concrete operator *op* to define the soundness and precision losses.

Learning Problem. Given a dataset $D = \{X_i\}$ representing some set of possible inputs to the abstract transformer $\hat{op}$ (for concrete operator *op*) in some abstract domain $\mathcal{A}$, we pose the learning of the neural abstract transformer $\hat{op}^*$ in an unsupervised manner as the following optimization problem:

$$\min_{\theta} \mathbb{E}_{X_i \sim D} \left[\alpha * \mathcal{L}''_S(op,\ X_i,\ \hat{op}^*(X_i; \theta)) + \beta * \mathcal{L}''_P(\hat{op}^*(X_i; \theta))\right] \tag{3.3}$$

which is based on the following components:

1. **Soundness Loss.** The primary challenge here is the absence of ground truth outputs from the transformer for calculating the soundness loss. For abstract states $a_1, a_2 \in \mathcal{A}$, state a_2 is considered a sound abstract representation of the effects of operation *op* on the concrete states denoted by a_1 if it over-approximates the effect of *op* on a_1, formally expressed as $op(\gamma(a_1)) \sqsubseteq_C \gamma(a_2)$ (Eq 2.4). Consequently, an effective metric for soundness loss, $\mathcal{L}''_S(op, a_1, a_2)$, should return 0 if $op(\gamma(a_1)) \sqsubseteq_C \gamma(a_2)$ holds true and otherwise return a positive value that approximates the size of the set $op(\gamma(a_1)) \setminus \gamma(a_2)$, so that it can be used to guide the network towards a sound transformer. An issue that

arises here is that, in most cases, the check $op(\gamma(a_1)) \sqsubseteq_C \gamma(a_2)$ can only be done by encoding the semantics of op (and γ) in SMT and then using an SMT solver. However, SMT solving is not differentiable and cannot guide the learning.

To counter the issues discussed above, we first introduce the notion of *distance* $\mathcal{D}(c, a)$ between a point in the concrete domain c and a point in the abstract domain a. This defines how "far" is the point c from being included in the set of concrete points that a represents ($\gamma(a)$). For example, for the interval domain, if $a = [2, 4]$ and the $c = 7$, $\mathcal{D}(c, a) = 7 - 4 = 3$. Also, note that $\mathcal{D}(c, a) = 0$ if $c \in \gamma(a)$.

Now, we define the notion of *maximum violating concrete point (MVCP).*

Definition 3.1. For a tuple (op, a_1, a_2), where op is some operator in the concrete domain and a_1, a_2 are points in the abstract domain $\mathcal{A}$, a maximum violating concrete point or MVCP is a state $c_m \in \mathcal{C}$ that belongs to $op(\gamma(a_1))$ but is not contained in the concretization of a_2 and is the *farthest* from a_2, where the notion of farthest is defined using the notion of $\mathcal{D}(c, a_2)$ described above. If the set $op(\gamma(a_1)) \setminus \gamma(a_2)$ is empty (which means a_2 is a sound output for op on a_1), then there is no MVCP for the tuple (op, a_1, a_2).

Mathematically, MVCP for a tuple (op, a_1, a_2) is defined as the following:

$$\begin{aligned} \underset{c}{\text{argmax}} \quad & \mathcal{D}(c, a_2) \\ \text{subject to} \quad & c \in \text{op}(\gamma(a_1)) \setminus \gamma(a_2) \end{aligned} \tag{3.4}$$

Now, using the notion of $MVCP(op, a_1, a_2)$ and $\mathcal{D}(c, a)$, we define $\mathcal{L}''_S$ to be:

$$\mathcal{L}''_S(op, a_1, a_2) = \mathcal{D}(MVCP(op, a_1, a_2),\ a_2) \tag{3.5}$$

While using this loss in Eq 3.3, a_1 is the input abstract state (X_i), and a_2 is the abstract state given by the neural transformer ($\hat{op}^*(X_i; \theta)$). Intuitively, minimizing this loss helps learning sound transformer in the following way:

(a) $MVCP(op,\ X_i,\ \hat{op}^*(X_i; \theta))$ returns the most distant concrete state that is missing in the concretization of $\hat{op}^*(X_i; \theta)$ but should be included. In Fig 3.1, the $MVCP$ is shown as the green dot and the concretization of $\hat{op}^*(X_i; \theta)$ is displayed by the red region. As can be seen, the green dot is the furthest point in the green region (which means it should be in concretization of the output), which is not in the red region.

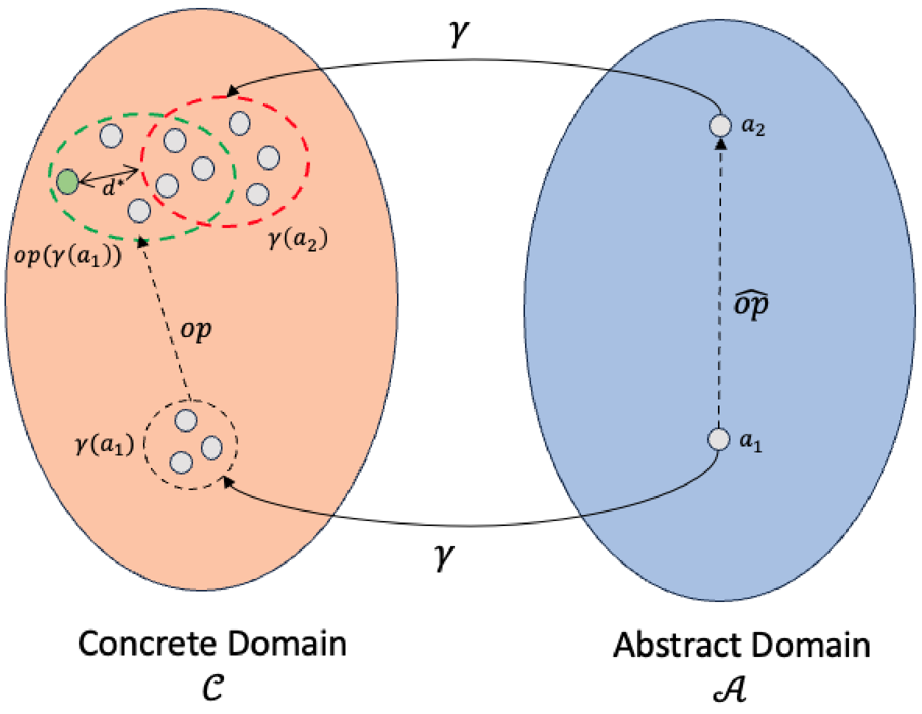


Figure 3.1: Here, a_2 does not constitute a sound output on a_1 with respect to op due to the fact that $op(\gamma(a_1)) \not\sqsubseteq_C \gamma(a_2)$, as demonstrated by the green and red boundary regions. Consequently, to guide the model toward learning sound outputs, the green dot, which is the *maximum violating concrete point* (MVCP) and lies outside $\gamma(a_2)$ at the greatest distance, is identified. The loss is defined by the distance d^* between the MVCP and $\gamma(a_2)$; minimizing this loss promotes the inclusion of the MVCP within $\gamma(a_2)$, thereby guiding the model towards sound transformer outputs.

(b) Once a point c_m is recognized as the MVCP, minimizing distance $\mathcal{D}(c_m,\ \hat{op}^*(X_i;\theta))$ (represented by d^* in Fig 3.1) guides the model towards outputs whose concretizations include c_m. This guides the model toward sound transformers, as at each iteration, the model tries to decrease the distance between the output concretization and the MVCP, thus converging to models where this distance is mostly zero. When d^* is 0, it means that the transformer is sound as there is no MVCP.

Example. Consider a case where we are trying to learn a neural transformer $\hat{abs}^*$ in the Interval domain for the *abs* operator. Say input $a_1 = [-10, 15]$ and the neural transformer returns $a_2 = [0, 12]$ for a_1. It is easy to check that $abs(\gamma(a_1))$ is given by $[0, 15]$ and so any sound output should contain $[0, 15]$. As $[0, 15]$ is not contained in $a_2 = [0, 12]$, we find the farthest point from $[0, 12]$ that should be included in it and get the MVCP as $c_m = 15$. This is now used to compute soundness loss $\mathcal{L}''_S(c_m, a_2) =$

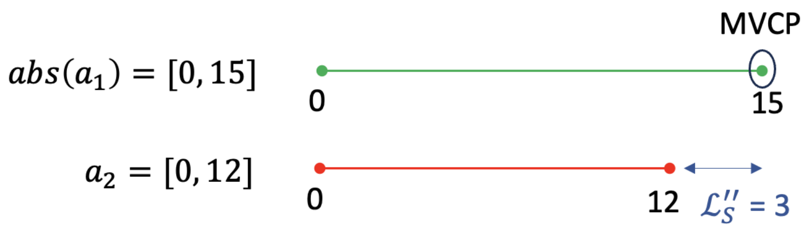


Figure 3.2: While learning neural transformer for abs, when a_1 is $[-10, 15]$, the output should contain $[0, 15]$ for soundness. So, a_2 is not sound, and the MVCP $c_m = 15$ is used to compute the soundness loss.

$\mathcal{L}''_S(15, [0, 12]) = 15 - 12 = 3$. This $\mathcal{L}''_S$ then guides the model to transformers that output sound results for a_1.

Note that we still need to use SMT solvers to compute the MVCP, but once it is computed, the distance function $\mathcal{D}$ (which can be differentiably implemented for many domains, as discussed in the next sections) is used to guide the learning of the network.

2. **Precision Loss.** Consider a measure $\mathcal{M}(a)$ of the size of the set $\gamma(a)$ abstracted by an element $a \in \mathcal{A}$. In the unsupervised scenario, where ground truth outputs for abstract transformers are not available, we enforce precision by directly utilizing the size $\mathcal{M}(a)$ of the model output a. More precisely, $\mathcal{L}''_P(a)(\geq 0)$ provides a differentiable approximation of the size $\mathcal{M}(a)$. For example, for the Interval domain, the size of an abstract element $[l, u]$ is given by $\mathcal{M}([l, u]) = \max(u - l, 0)$, which can be minimized to enforce precision. Minimizing $\mathcal{L}''_P(\hat{op}^*(X_i; \theta))$ guides the model towards learning neural transformers that generate smaller outputs.

3. **Soundness & Precision Weights.** As in the case of supervised learning, α, β are the soundness and precision weights, respectively, and let us control the degree of soundness and precision required for the transformer. This can be adjusted based on the specific downstream analysis task for which the neural transformer will be used.

As we will demonstrate in the next sections, the SMT query to find $MVCP(op, a_1, a_2)$ can be easily implemented for various operators in the Interval and Octagon domain. Also, the functions $\mathcal{D}(c, a)$ and $\mathcal{L}''_P(a)$ can be computed easily and differentiably for these domains.

Datasets. For the unsupervised learning setting, the datasets can be generated by randomly sampling abstract elements from the abstract domain $\mathcal{A}$.

Benefits. Apart from learning neural transformers that are faster and allow differentiable analysis, the unsupervised learning of neural transformers has the following added benefits:

1. It does not require a hand-crafted transformer and allows for the automatic generation of abstract transformers. It requires as inputs the domain-related functions $\mathcal{D}(c, a)$ and $\mathcal{L}''_P(a)$ and the semantics of $MVCP(op, a_1, a_2)$, where *op* is the operator for which the abstract transformer is to be learned. Once these are specified, the soundness and precision weights (α and β) can be tweaked to generate transformers with varying soundness and precision automatically.

2. For certain operations, like affine assignment in the octagon domain, it's challenging to implement efficient abstract transformers without losing precision. However, as neural networks can approximate complex functions, the unsupervised learning method can guide the network to learn efficient and more precise transformers for such operations.

3.5 SOUNDNESS PRECISION TRADE-OFF

In both supervised and unsupervised learning methods, there are two loss components: one ensuring soundness and the other ensuring precision. Solely having a soundness loss can guide the model toward very large outputs, which may be very imprecise for use in downstream tasks. This necessitates the use of precision loss to keep the model outputs precise. However, this results in a trade-off. The soundness loss tries to enforce soundness by increasing the size of the output element to include the sound region. On the other hand, the precision loss guides the model towards learning outputs with smaller sizes to maintain precision. This makes our learning problems multi-objective learning problems with conflicting objectives, thus making them hard. By appropriately tuning the values of the soundness and precision weights (α and β), neural transformers can be learned with varying degrees of soundness and precision. In settings such as verification, where soundness is very important, we can always resort to hand-crafted transformers' outputs if the output of the neural transformer is unsound (note that checking the soundness of an output is comparably simpler).

CHAPTER 4: INSTANTIATION FOR NUMERICAL DOMAINS

In this section, we instantiate our `NeurAbs` framework for two widely used numerical domains: Interval and Octagon, and show how neural abstract transformers can be learned for operators in these domains.

4.1 INTERVAL DOMAIN

In the Interval domain [Sec 2.4.1], the abstract element is an interval. For example, the set $\{1.1, 2.2, 3.15, -1.3\}$ can be abstracted using the interval $[-1.3, 3.15]$. All possible values for a variable x in a program can be represented by an interval $[a, b]$. An element in the Interval domain is represented by two reals: l and u, where l represents the lower bound of the interval and u represents the upper bound of the interval.

4.1.1 Tensor Representation of Intervals

An element in the Interval domain can be represented as a tensor of size 2, where the first element represents the lower bound l of the interval and the second element represents the upper bound u of the interval. For example, the interval $[2.1, 3.5]$ can be represented as tensor([2.1, 3.5]). Thus, neural transformers for operators that take n intervals as inputs will have 2*n inputs and two outputs (representing the output interval). For example, the neural transformer to learn interval join (which takes two intervals and returns their join) will have four inputs l_1, u_1, l_2, u_2 and will output two numbers l_o, u_o which represent the output interval $[l_o, u_o]$.

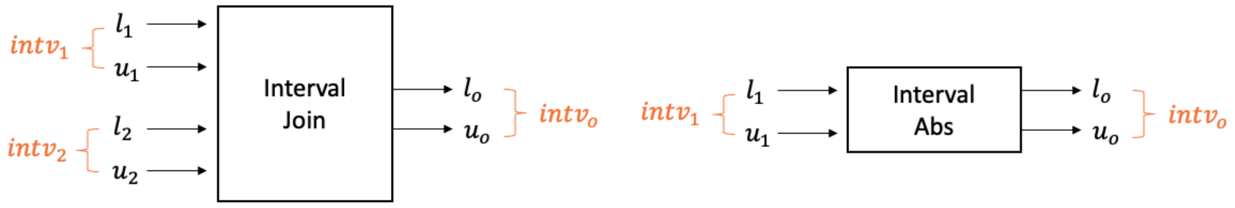


Figure 4.1: Neural Transformers for Interval Join and Interval Abs

4.1.2 Supervised Learning of Neural Interval Transformers

The general supervised learning approach described in Section 3.3 can be used to learn neural transformer for the Interval domain by using the following instantiations of the loss components present in Eq. 3.2:

1. $\mathcal{L}'_S(intv_1,\ intv_2)$: Given intervals $intv_1 = [l_1, u_1]$ and $intv_2 = [l_2, u_2]$, this loss has to enforce that $intv_2$ over-approximates $intv_1$, i.e. $intv_2$ contains all the points present in $intv_1$. This is only possible if $l_2 \leq l_1$ and $u_2 \geq u_1$. To enforce this, we need to penalize the model whenever $l_2 > l_1$ or $u_2 < u_1$. This can be enforced by defining $\mathcal{L}'_S([l_1, u_1],\ [l_2, u_2])$ as following:

$$\mathcal{L}'_S([l_1, u_1],\ [l_2, u_2]) = max(l_2 - l_1, 0) + max(u_1 - u_2, 0) \tag{4.1}$$

   Note that this loss is always ≥ 0, and will guide the model to decrease l_2 and increase u_2 when it is more than 0. It is 0 iff $l_2 \leq l_1$ and $u_2 \geq u_1$, which are the required conditions for soundness.

2. $\mathcal{L}'_P(intv_1,\ intv_2)$: Given intervals $intv_1 = [l_1, u_1]$ and $intv_2 = [l_2, u_2]$, this loss has to enforce that the size of $intv_2$ is close to size of $intv_1$. For this, we define the measure of the size of the interval $[l, u]$ as $\mathcal{M}([l, u]) = \max(u - l, 0)$. We take the max with 0 as the interval represented by $[l, u]$ is empty if $l > u$. Now, using this measure, the precision condition can be enforced by defining $\mathcal{L}'_P([l_1, u_1],\ [l_2, u_2])$ as follows:

$$\begin{aligned} \mathcal{L}'_P([l_1, u_1],\ [l_2, u_2]) &= \max(\mathcal{M}([l_2, u_2]) - \mathcal{M}([l_1, u_1]), 0) \\ &= \max(\max(u_2 - l_2, 0) - \max(u_1 - l_1, 0), 0) \end{aligned} \tag{4.2}$$

   Minimizing $\mathcal{L}'_P(intv_1,\ intv_2)$ guides the model towards outputting intervals that are closer to the size of the original intervals, thus maintaining precision.

4.1.3 Unsupervised Learning of Neural Interval Transformers

The general unsupervised learning approach described in Section 3.4 can be used to learn neural transformer for the Interval domain by using the following instantiations of the loss components present in Eq. 3.3:

1. $\mathcal{L}''_S(op,\ intv_1,\ intv_2)$: As described above, $\mathcal{L}''_S$ depends on the definitions of $\mathcal{D}(c, a)$ and $MVCP(op, a_1, a_2)$, where c is some element in the concrete domain $\mathcal{C}$ and a_1, a_2 belong to the abstract domain $\mathcal{A}$.

For the Interval domain, $\mathcal{D}(c, [l, u])$ where $c \in \mathbb{R}$, can be defined as:

$$\mathcal{D}(c, [l, u]) = \begin{cases} l - c & \text{if } c < l \\ c - u & \text{if } u < c \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

$\mathcal{D}(c, [l, u])$ captures how "far" is c from $[l, u]$.

Next, we define $MVCP(op, a_1, a_2)$ for the abs and the $join$ operator:

(a) **abs**: $MVCP(abs, [l_1, u_1], [l_2, u_2])$ should return $c \in \mathcal{R}$ that is present in $abs([l_1, u_1])$ and is farthest from $[l_2, u_2]$. This can be found by solving the following optimization problem:

$$\begin{aligned} \underset{c}{argmax} \quad & \mathcal{D}(c, [l_2, u_2]) \\ \text{subject to} \quad & \exists x.\ l_1 \leq x \leq u_1 \ \wedge \ c = |x| \end{aligned} \tag{4.4}$$

(b) **join**: $MVCP(join, ([l_1, u_1],\ [l_2, u_2]),\ [l_3, u_3])$ should return $c \in \mathcal{R}$ that is present in the join of $[l_1, u_1]$ and $[l_2, u_2]$ (i.e., it is present in one of those intervals) and is farthest from $[l_3, u_3]$. This can be found by solving the following optimization problem:

$$\begin{aligned} \underset{c}{argmax} \quad & \mathcal{D}(c, [l_3, u_3]) \\ \text{subject to} \quad & (l_1 \leq c \leq u_1) \ \vee \ (l_2 \leq c \leq u_2) \end{aligned} \tag{4.5}$$

The optimization procedures described above can be directly encoded in an SMT solver to get the MVCPs. $\mathcal{L}''_S(op,\ intv_1,\ intv_2)$ can be then implemented using these MVCPs and $\mathcal{D}$ defined above using the formulation in Eq. 3.5.

2. $\mathcal{L}''_P(intv)$: We define the measure of the size of the interval $[l, u]$ as $\mathcal{M}([l, u]) = \max(u - l, 0)$. We then use this measure directly to define $\mathcal{L}''_P([l, u])$ as follows

$$\mathcal{L}''_P([l, u]) = \mathcal{M}([l, u]) = \max(u - l, 0) \tag{4.6}$$

Minimizing this would guide the model towards outputting smaller intervals, thus maintaining precision.

4.2 OCTAGON DOMAIN

In the Octagon domain [Sec 2.4.2], the possible values are abstracted using the *octagon shape*. If the program has n variables $v_1, v_2, ...v_n$, then the octagonal representation of the program state is given by constraints of the form $\pm v_i \pm v_j \leq c_{ij}$ and $\pm v_i \leq d_i$.

4.2.1 Tensor Representation of Octagons

To begin learning neural transformers for the octagon domain, it is essential first to convert an octagon domain element into a tensor. We can represent the octagonal constraints as an array/tensor consisting only of the inequality constants (c_{ij} and d_i values) if we can define some order on all the possible constraints. Within the `NeurAbs` framework, we use the following ordering on the octagon constraints to encode an octagon as a tensor:

1. We first capture constraints that are on just one variable. Assuming we have an ordering on variables as above, the first constraint would correspond to $v_1 <= c_1$, the second would correspond to $-v_1 <= c_2$, and so on. This amounts to $2 * n$ values (2 for each variable).

2. Next, we capture the constraints between two variables. For this, we traverse over the possible pair of variables in the following order: $[(v_1, v_2),\ (v_1, v_3)\ (v_1, v_4)\ \ldots\ (v_1, v_n),$ $(v_2, v_3), (v_2, v_4), \ldots (v_{n-1}, v_n)]$. For a pair (v_i, v_j), we use the following order for possible constraints: $[(v_i + v_j),\ (v_i - v_j),\ (-v_i + v_j),\ (-v_i - v_j)]$

   The number of possible variable pairs is $n*(n-1)/2$, and there are 4 possible constraints for each pair. This amounts to $4 * n * (n - 1)/2 = 2 * n * (n - 1)$ values.

In total, these leads to $2 * n + 2 * n * (n - 1) = 2n^2$ inequality constants for an octagon with n variables. We denote these inequality constants by w_i (for the i^{th} inequality in the above-defined order). However, it is not necessary to have all the inequalities to define the octagon. For instance, the equation $x - y \leq 1$ is also a valid octagon representation. In this case, we do not have the inequality constants for other inequalities like $x + y$ or x. To tackle this, we also use $2 * n^2$ inequality indicator variables $e_i = \{0, 1\}$ that indicate if the i^{th} inequality is present in the octagon representation. This final representation then has $4 * n^2$ values $([w_1, w_2, \ldots w_n, e_1, e_2, \ldots e_n])$. We will use $[w, e]$ as a short-hand notation to denote octagons, where w and e represent the constants part and the indicators part of the octagon tensor. For the inequalities that are not present, we use a high constant $(\mathcal{K})$ as a proxy for the inequality constant. Technically, the absence of the inequality implies that it is less than ∞, and therefore, using a high constant as a substitute is a sensible choice.

Example. Say we have 2 variables in our program: x and y (and we fix this order). Consider the octagon $O = \{x \leq 5, -y \leq 2, x+y \leq 10, -x+y \leq 20\}$. All possible constraints in the order defined above would be $[x, -x, y, -y, x+y, x-y, -x+y, -x-y]$. Corresponding to this, the inequality constants part of the tensor would look like $w = [5, \mathcal{K}, \mathcal{K}, 2, 10, \mathcal{K}, 20, \mathcal{K}]$ and the inequality indicator part would be $e = [1, 0, 0, 1, 1, 0, 1, 0]$. The final tensor representation would be a concatenation of w and i as $[w, e] = [5, \mathcal{K}, \mathcal{K}, 2, 10, \mathcal{K}, 20, \mathcal{K}, 1, 0, 0, 1, 1, 0, 1, 0]$

The tensor representation described above captures all the details of the octagons and can now be passed to neural networks. For example, a neural transformer to learn octagon join for octagon with n variables each will take 2 octagons as inputs. This means that it will have $8 * n^2$ inputs ($4 * n^2$ for each octagon) and will return $4 * n^2$ outputs that represent the output octagon.

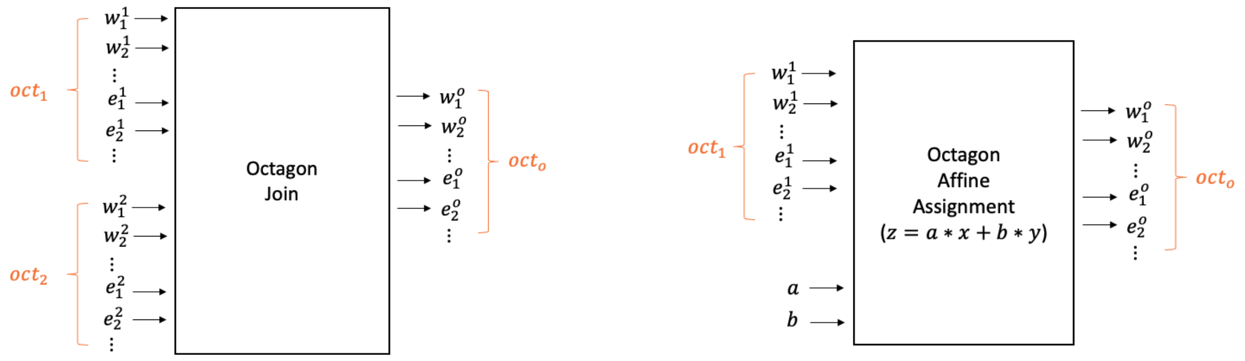


Figure 4.2: Neural Transformers for Octagon Join and Affine Assignment

The neural networks for octagon transformers apply sigmoid to the indicator outputs e. So, the e values returned by the neural octagon transformers are always in the range of 0 to 1. A suitable threshold (like ≥ 0.5) can then be applied to choose or not choose the inequality in the resultant octagon. If the i^{th} inequality is chosen, its weight is given by the output w_i.

4.2.2 Supervised Learning of Neural Octagon Transformers

The general supervised learning approach described in Section 3.3 can be used to learn neural transformer for the Octagon domain by using the following instantiations of the loss components present in Eq. 3.2:

1. $\mathcal{L}'_S(oct_1,\ oct_2)$: Let octagon oct_1 be represented as $oct_1 = [w, e]$ and oct_2 be represented as $oct_2 = [w', e']$. Note that while using $\mathcal{L}'_S$, oct_1 is the ground truth, and so its

indicators will be $\{0, 1\}$ while oct_2 is the output from the neural network, and its indicators will be $0 \leq e'_i \leq 1$. As described above, the i^{th} inequality is included in the output octagon if $e'_i \geq 0.5$.

Now, this loss has to enforce that oct_2 over-approximates oct_1, i.e., oct_2 should have all the points covered by oct_1. To enforce this, we enforce that no possible inequalities are *stricter* in oct_2 as compared to oct_1. Equality i is stricter in oct_2 if one of the following holds:

(a) $e_i = 1$ and $e'_i \geq 0.5$ and $w'_i < w_i$ (both octagons have inequality i but the constant is less for oct_2).

(b) $e_i = 0$ and $e'_i \geq 0.5$ (Only oct_2 has the i^{th} inequality).

Note that oct_2 can over-approximate oct_1 even if it has some inequalities that are stricter than those in oct_1, as the other inequalities can compensate for this. Therefore, it is not a necessary condition. However, it is easy to verify that ensuring no inequalities in oct_2 are stricter provides a sufficient condition for soundness (proved in Appendix A.2). To enforce this, we first define the contribution of the i^{th} inequality to soundness loss as follows:

$$l_i = \begin{cases} k_1 * (w_i - w'_i) & \text{if } e'_i \geq 0.5 \text{ and } e_i = 1 \text{ and } w_i > w'_i \\ k_2 * BCELoss(e'_i, e_i) & \text{if } e'_i \geq 0.5 \text{ and } e_i = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Here, k_1 and k_2 are constants and can be chosen appropriately. BCELoss is the BinaryCrossEntropy Loss. The loss defined above penalizes the model whenever it sees one of the two conditions for i^{th} inequality being stricter in oct_2. In the first case, the model is penalized if i^{th} inequality is present in both octagons, but oct_2 has a smaller constant for it. In the second case (only oct_2 has the i^{th} inequality), the loss guides the output model to have e'_i closer to $e_i = 0$, thus learning models that do not have the i^{th} inequality in their outputs, which eventually ensures soundness.

$\mathcal{L}'_S(oct_1,\ oct_2)$ can be computed by taking the mean of l_is for all possible constraints, i.e. $(\sum_{i=1}^{N} l_i)/N$, where $N = 2 * n^2$ and n is the number of variables in the octagon.

2. $\mathcal{L}'_P(oct_1,\ oct_2)$: Given two octagons oct_1 and oct_2, $\mathcal{L}'_P(oct_1,\ oct_2)$ needs to enforce that they are close in size. For this, say there is a measure of the size of octagon $\mathcal{M}(oct)$. $\mathcal{L}'_P(oct_1,\ oct_2)$ should then return a differentiable approximation of the difference in two measures $(M(oct_2) - M(oct_1))$. Octagons are polytopes (can be unbounded also), and it

is, in general, difficult to come up with a measure for the size of the octagon. However, we can approximate how big octagon $oct_2 = [w', e']$ is as compared to $oct_1 = [w, e]$ using the following:

(a) Number of inequality constraints present in oct_1 that are not in oct_2 (i.e. $\{i \mid e'_i < 0.5 \ \wedge \ e_i = 1\}$). In most cases, a smaller number of inequalities means that the octagon covers a larger area.

(b) Cases where the inequality constants are larger in oct_2. As the inequalities are of $\leq$ form, a higher inequality constant means that the inequality satisfies more number of points.

The two metrics above can be combined to define $\mathcal{L}'_P([w, e],\ [w', e'])$ as:

$$k_1 * BCELoss(e'_i, e_i \mid e_i = 1) + k_2 * \sum_i (max(w'_i - w_i,\ 0)) \tag{4.8}$$

Here, k_1 and k_2 are constants and can be chosen appropriately. $BCELoss(e'_i, e_i \mid e_i = 1)$ means that the BinaryCrossEntropy loss is only computed for inequality i if $e_i = 1$, i.e. it is present in the first octagon. Minimizing $\mathcal{L}'_P(oct_1,\ oct_2)$ guides the model towards octagons oct_2 which are relatively similar to the size of octagons oct_1 by adding inequalities not present in oct_2 (but present in oct_1) and decreasing inequality constants in oct_2.

4.2.3 Unsupervised Learning of Neural Octagon Transformers

The general unsupervised learning approach described in Section 3.4 can be used to learn neural transformer for the Octagon domain by using the following instantiations of the loss components present in Eq. 3.3:

1. $\mathcal{L}''_S(op,\ oct_1,\ oct_2)$: As described above, $\mathcal{L}''_S$ depends on the definitions of $\mathcal{D}(c, a)$ and $MVCP(op, a_1, a_2)$, where c is some element in the concrete domain $\mathcal{C}$ and a_1, a_2 belong to the abstract domain $\mathcal{A}$.

   For the Octagon domain, we will first define $\mathcal{D}(c, oct)$. For the octagon $oct = [w, e]$, let $v_i(c)$ be the value of the i^{th} possible inequality expression on the point c. For instance, if the i^{th} inequality is on $x + y$ and we have $c = \{x : 2, y : 3\}$, $v_i(c) = 5$. We then collect the set of inequalities of oct not satisfied as c, which is given by the set of indices

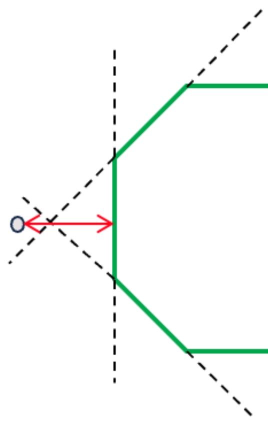


Figure 4.3: The dotted lines show the inequalities the point does not satisfy. The distance in red shows the $\mathcal{D}(c, oct)$.

$\delta(c, [w, e])$ defined as:

$$\delta(c, [w, e]) = \{i \mid e_i = 1 \ \wedge \ v_i(c) > w_i\} \tag{4.9}$$

We define $\mathcal{D}(c, oct)$ as the maximum of the distances of c from the inequalities that c does not satisfy (given by $\delta(c, [w, e])$), i.e.

$$\mathcal{D}(c, [w, e]) = \underset{i \in \delta(c,[w,e])}{max} v_i(c) - w_i \tag{4.10}$$

Thus, $\mathcal{D}(c, oct)$ measures the distance of c from the inequality that is *most-violated*. Note that $\mathcal{D}(c, oct)$ is defined to be 0 if $c \in oct$. Once $\mathcal{D}(c, oct)$ is defined, we need to find the $MVCP$. As discussed earlier, $MVCP$s are computed by encoding the $MVCP$ constraints into SMT solvers. Before discussing the query used to find the $MVCP$s, we first define $encode(oct, [v_1, v_2, \ldots v_n])$ as the encoding on a n variable octagon using symbolic variables $v_1, v_2, \ldots v_n$. This can be done easily by asserting the inequalities in the octagon oct on the specified variables. For example, say the octagon is $\{x + y \geq 20, x < 2\}$. This can be encoded using symbolic variables $[v_1, v_2]$ as $(v_1 + v_2 \geq 20 \ \wedge \ v_1 < 2)$.

Now, consider the case of octagons with 2 variables and the affine assignment operator that finds the new octagon as a result of the statement $x = a * x + b * y$. $MVCP(op, oct_1, oct_2)$ for this operator (affine assignment) can be computed using:

$$\begin{aligned} \underset{c}{argmax} \quad & \mathcal{D}(c, oct_2) \\ \text{subject to} \quad & \exists v_1, v_2.\ encode(oct_1, [v_1, v_2]) \ \wedge c = (a * v_1 + b * v_2, v_2) \end{aligned} \tag{4.11}$$

Here, $encode(oct_1, [v_1, v_2])$ encodes that v_1 and v_2 belong to the octagon oct_1. Under this condition, we find the point $c = (a * v_1 + b * v_2, v_2)$ that should be in the resultant octagon (after the affine operation) but is the farthest from oct_2.

Similarly, $MVCP(join, (oct_1, oct_2), (oct_o))$ for octagon join (3 variables octagons) can be computed using:

$$\begin{aligned} \underset{(v_1,v_2,v_3)}{argmax} \quad & \mathcal{D}([v_1, v_2, v_3], oct_o) \\ \text{subject to} \quad & encode(oct_1, [v_1, v_2, v_3]) \ \vee encode(oct_2, [v_1, v_2, v_3]) \end{aligned} \tag{4.12}$$

In this case, we try to find the concrete point farthest from oct_o that is present in at least one of oct_1 or oct_2 (and so should be in the join).

$\mathcal{L}''_S(op,\ oct_1,\ oct_2)$ can be then implemented using the $MVCP$s and $\mathcal{D}$ defined above using the formulation in Eq. 3.5. Reducing the distance $\mathcal{D}$ of the $MVCP$s from the model's output oct_2 guides the model towards sound transformers.

2. $\mathcal{L}''_P(oct)$: $\mathcal{L}''_P(oct)$ ensures that the learned octagons are smaller in size and thus enforce precision. For this, $\mathcal{L}''_P(oct)$ should return a differentiable approximation of some measure $\mathcal{M}$ of the size of the octagon. However, as discussed earlier, defining such a measure for octagons (which are polytopes and can also be unbounded) is not trivial. Instead, we rely on these two metrics to approximate the size of an octagon:

   (a) Number of inequalities in the octagon: If an octagon has fewer inequalities, it usually means that it covers a large area. Thus, it makes sense to enforce that the produced octagons have as many inequalities as possible to enforce precision.

   (b) Inequality constants of the inequalities present: If the inequality constants of the inequalities in the octagon are higher, it usually means that it covers a larger area (as the inequalities are of $\leq$ type). Thus, it makes sense to enforce that the inequality constants in the octagons are smaller in value.

The above two metrics can be combined to define $\mathcal{L}''_P([w, e]])$ as follows:

$$k_1 * BCELoss(e_i, 1 \mid e_i < 0.5) + k_2 * \sum_{i \mid e_i \geq 0.5} w_i \tag{4.13}$$

Here, k_1 and k_2 are constants and can be chosen appropriately. $BCELoss(e, 1 \mid e < 0.5)$ means that the BinaryCrossEntropy loss should be used only for inequalities i not present in the octagon ($e_i < 0.5$). This guides the model towards learning octagons with more inequalities by pushing e_is, which are less than 0.5, towards 1. The second term $\sum_{i \mid e_i \geq 0.5} w_i$ denotes the sum of inequality constants of those inequalities which are present in the octagon ($e_i \geq 0.5$). This term guides the model towards learning octagons with smaller inequality constants. These two components together allow $\mathcal{L}''_P([w, e]])$ to enforce precision and guide the model towards smaller octagons.

CHAPTER 5: EVALUATION

In this section, we demonstrate the capability of the `NeurAbs` framework to train transformers that are both sound and precise for the Interval and Octagon domains. Additionally, we explore how the differentiability of neural transformers enables the formulation of tasks such as invariant generation as learning problems.

5.1 NEURAL INTERVAL TRANSFORMERS: SOUNDNESS & PRECISION

In this section, we evaluate the efficacy of the `NeurAbs` framework to learn neural abstract transformers for the Interval domain. We pick two abstract transformers, abs and join, described in Section 2.4.1.

Datasets. For supervised learning, we gather data for each abstract transformer by generating random input intervals and using existing hand-crafted transformers to produce the corresponding ground truth outputs. For training and testing purposes, we create 10,000 input-output pairs for each abstract transformer.

5.1.1 Supervised Learning of Interval Transformers

We use the loss methods described in Section 4.1.2 to train neural interval transformers for abs and join in a supervised manner by training on 5000 samples each. The results are displayed in Table 5.1. The first column indicates the soundness and precision weights (α, β) used while training the networks. The first row in the table (with $-$ for α and β) depicts the performance of a random neural network. To evaluate the quality of the learned transformers, we evaluate two metrics:

1. Soundness Measure (%): This is the percentage of sound outputs generated by the neural transformer when evaluated on a test set with 10,000 data points.
2. Imprecision Measure: For the cases where the outputs were sound, this measures how big the output intervals were as compared to the ground truth intervals. This is computed as the sum of differences in sizes of output and ground truth intervals for the sound cases divided by the number of sound cases.

As can be seen, randomly initialized neural networks are ineffective as neural transformers because they are very unsound. The results display that our supervised learning approach

Table 5.1: Neural Interval Transformers for Abs and Join trained using supervised learning method.

Weights (α, β)	Interval Abs		Interval Join	
(Soundness, Precision)	Soundness (%)	Imprecision	Soundness (%)	Imprecision
(-, -)	20.03	4.44	3.88	0.16
(1, 1)	26.39	1.57	32.34	40.16
(2, 1)	47.43	5.74	40.53	25.70
(5, 1)	66.88	11.70	63.10	43.58
(7, 1)	84.02	10.39	73.07	113.78
(10, 1)	97.72	18.41	89.24	116.31
(50, 1)	99.99	40.63	99.57	191.72

allows us to learn sound and precise neural transformers. As expected, increasing the soundness weight guides the model to learn more sound transformers, but this makes the model less precise. This also shows the effectiveness of our framework in learning multiple neural transformers for the same abstract transformer with varying soundness and precision.

5.1.2 Unsupervised Learning of Interval Transformers

We use loss methods described in Section 4.1.3 to train neural interval transformers for the abs and join operations through unsupervised learning, using 1000 samples each. We opted for a smaller dataset because unsupervised learning requires the computation of MVCPs for each training example in every iteration, a process that involves computationally intensive calls to an SMT solver. Despite the reduced number of training data points, the results presented in Table 5.2 confirm that our training method successfully learns sound and precise transformers. Also, as the model does not have access to the ground truth outputs in the unsupervised learning approach, it is easier for it to “sway” towards models that are sound but very imprecise. Thus, a higher precision weight (10 here) is used to achieve similar results to the supervised learning method.

As in the previous section, the first column indicates the soundness and precision weights (α, β) used while training the networks, and the first row depicts the performance of a random neural network. The soundness and the imprecision measures are also the same as described above. Note that, while computing the precision loss in the unsupervised learning approach, we only use the size of the the output intervals to enforce precision. However, to evaluate the quality of the learned transformers, we compute the difference in interval sizes of the ground truth and the learned transformers.

Table 5.2: Neural Interval Transformers for Abs and Join trained using the unsupervised learning method.

Weights (α, β)	Interval Abs		Interval Join	
(Soundness, Precision)	Soundness (%)	Imprecision	Soundness (%)	Imprecision
(-, -)	20.03	4.44	3.91	26.80
(20, 10)	25.04	4.29	38.99	164.61
(30, 10)	63.04	25.86	53.65	219.08
(50, 10)	85.96	36.95	93.03	255.93
(75, 10)	100	73.17	97.95	277.70

The results in Table 5.2 show the efficacy of our unsupervised learning method to learn sound and precise neural transformers for the interval domain. The soundness-precision trade-off is also evident as increasing the soundness weight generates models that are more sound but less precise.

5.2 NEURAL OCTAGON TRANSFORMERS: SOUNDNESS & PRECISION

In this section, we evaluate the efficacy of the `NeurAbs` framework to learn neural abstract transformers for the Octagon domain. Note that this is, in general, more difficult than the Interval domain because of the complex tensor representation of octagons and their more complex abstract transformers.

Datasets. For supervised learning, we gather data for the octagon join transformer (octagons with 3 variables) by generating random input intervals and using existing hand-crafted transformers to produce the corresponding ground truth outputs. For training and testing purposes, we create 10,000 input-output pairs for each abstract transformer.

5.2.1 Supervised Learning of Octagon Transformers

We use the loss methods described in Section 4.2.2 to train neural interval transformers for octagon join (octagons with 3 variables) in a supervised manner by training on 10,000 samples. The results are displayed in Table 5.3. The first column indicates the soundness and precision weights (α, β) used while training the networks. The first row in the table (with $-$ for α and β) depicts the performance of a random neural network. To evaluate the quality of the learned transformers, we evaluate two metrics:

1. Soundness Measure (%): This is the percentage of sound outputs generated by the

Soundness Weight (α)	Precision Weight (β)	Soundness Measure (%)	Imprecision Measure
-	-	0.0	(-, -)
10	100	10.3	(0.087, 76.16)
20	100	25.2	(0.075, 88.89)
50	100	32.5	(0.181, 101.53)
100	100	46.4	(0.157, 113.70)
150	100	60.9	(0.323, 135.59)
450	100	72.0	(0.502, 154.60)
700	100	79.2	(0.963, 175.29)

Table 5.3: Soundness and Precision of Neural Octagon Join

neural transformer when evaluated on a test set with 1000 data points.

2. Imprecision Measure: For the cases where the outputs were sound, we measure two things:
   (a) The average difference in the number of inequalities in the ground truth and the transformer output. A higher difference indicates that the transformer output has fewer inequalities and, thus, is more imprecise.
   (b) The average difference in the sum of inequality constants for the transformer output and the ground truth. A higher difference indicates that the transformer output has higher inequality constants and, thus, is more imprecise.

As can be seen, the randomly initialized neural network is ineffective as a neural transformer because it is never sound. The results display that our supervised learning approach allows us to learn sound and precise neural transformers for a complex transformer like octagon join. As expected, increasing the soundness weight guides the model to learn more sound transformers, but this makes the model less precise. This relationship is evident from the table: an increase in soundness corresponds with a decrease in the number of inequalities in the output octagon and an increase in the constants of these inequalities as we move downward through the entries. These results also show the effectiveness of our framework in learning multiple neural transformers for the same abstract transformer with varying soundness and precision.

5.2.2 Unsupervised Learning of Octagon Transformers

To demonstrate the effectiveness of the unsupervised learning approach in the Octagon domain, we pick the affine assignment operator. Specifically, we consider octagons with 2

Soundness Weight (α)	Precision Weight (β)	Soundness Measure (%)	Imprecision Measure
-	-	0.0	(0, 0)
10	1000	1.5	(8.0, 1601.85)
100	1000	23.6	(4.2, 799.85)
450	1000	41.5	(3.1, 405.79)
550	1000	59.2	(2.0, 305.12)
600	1000	77.3	(1.0, 198.19)

Table 5.4: Soundness and Precision of Neural Octagon Affine Assignment ($x = a * x + b * y$)

variables (x and y) and consider affine assignments of the form $x = a * x + b * y$. The affine operator returns the resultant octagon after the affine assignment. For the dataset, we generate random octagons and random values of a and b. We do not need the ground truths as we will use the unsupervised learning method. We use loss methods described in Section 4.2.3 to train a neural transformer for the affine assignment operator through unsupervised learning using 1000 samples. We opt for a small number of data points as unsupervised learning requires the computation of MVCPs for each training example in every iteration, a process that involves computationally intensive calls to an SMT solver. However, our unsupervised learning method still enables the effective training of sound and precise transformers, as can be seen in Table 5.4.

As in the previous section, the first column indicates the soundness and precision weights (α, β) used while training the networks, and the first row depicts the performance of a random neural network. The soundness measure is the same as described above and depicts the percentage of sound outputs on a test dataset with 1000 data points. However, as we do not have the ground truth outputs in the unsupervised case, the imprecision measure computes the following for the cases where the output is sound:

1. The average number of inequalities present in the output octagon. A smaller number of inequalities means that the output octagon is less precise.
2. The average sum of inequalities present in the output octagon. A higher sum of inequalities indicates that the output octagons are less precise.

The data in Table 5.4 demonstrate the effectiveness of our unsupervised learning approach. As the soundness weight increases, there is a corresponding increase in the proportion of sound outputs produced. However, in line with the anticipated trade-off between soundness and precision, the precision of the output octagons decreases, as evidenced by a reduction in the number of inequalities they contain.

5.3 DIFFERENTIABLE LEARNING OF LOOP INVARIANTS

In this section, we highlight the advantages of our neural abstract transformers being differentiable by employing them in learning loop invariants. We frame this task of finding valid inductive octagonal invariants for a loop program $\mathcal{P}$ as a learning problem. Let's consider a typical loop program $\mathcal{P}$ = `while`(β) `do` $\mathcal{C}$ `od`. Let $\hat{\mathcal{C}} = \hat{op}_n \circ \ \hat{op}_{n-1} \cdots \circ \hat{op}_1$ represent the effective abstract transformer for $\mathcal{C}$ where $\hat{op}_i$ represents the abstract transformer for i^{th} statement in $\mathcal{C}$. If O_{init} approximates the initial states $(init)$ of $\mathcal{P}$, then the octagon O_{inv} is a valid octagonal invariant of the program if it satisfies:

$$(O_{init} \subseteq O_{inv}) \ \wedge \ (\hat{\mathcal{C}}(\hat{conj}(O_{inv}, \ \beta)) \subseteq O_{inv}) \tag{5.1}$$

where $\hat{conj}$ represents the abstract transformer for taking the conjunction of an octagon with a condition like β and so $\hat{conj}(O_{inv}, \ \beta)$ is an approximation for points in O_{inv} that satisfy β. Now, using our `NeurAbs` framework, we can derive the neural approximation $\hat{op}_i^*$ for each transformer $\hat{op}_i$. These neural transformers can then be composed to get the *effective neural transformer for the loop body* $(\hat{\mathcal{C}}^*)$ as $\hat{\mathcal{C}}^* = \hat{op}_n^* \circ \hat{op}_{n-1}^* ... \circ \hat{op}_1^*$. Exploiting the differentiability of $\hat{\mathcal{C}}^*$, we can now pose the search for a valid octagonal invariant O_{inv} by minimizing the following:

$$\mathcal{L}'_S(O_{init}, \ o) + \mathcal{L}'_S(\hat{\mathcal{C}}^*(\hat{conj}^*(o, \beta)), \ o) \tag{5.2}$$

where $\hat{conj}^*$ represents the neural transformer for conjunction. As defined in Section 4.2.2, $\mathcal{L}'_S(o_1, o_2)$ measures if o_2 over-approximates o_1 $(o_1 \subseteq o_2)$. This lets us enforce the condition that octagon o_1 is contained in octagon o_2.

Starting with random octagons, we can now apply gradient descent guided by the loss described in Eq. 5.2 to find candidate octagon invariants, which can then be validated by verifying the correctness constraints described in Eq. 5.1 using an SMT solver. This gradient-guided method provides an efficient way to guide the search for octagonal invariants.

```
x = 100;
y = 150;
while (y <= 600) {
    x = x + y;
    y = 2*y;
}
```

Figure 5.1: A loop program

As a concrete example, consider the loop program in Fig. 5.1. Here, the init condition $init$ given by $x = 100$; $y = 150$ can be represented by the octagon $O_{init} = \{x \geq 100, \ -x \leq -100, \ y \geq 150, \ -y \leq -150\}$. We first train a neural transformer for the affine assignment in the octagon domain. Starting with random octagons, we guide the search for valid octagonal invariants using the loss given by Eq. 5.2. Invariants are returned only after checking their validity using an SMT solver. This method helps us synthesize non-trivial valid octagonal invariants like:

1. $\{y \geq 65.514, x - y \leq -49.951, -x - y \leq 74.897\}$
2. $\{x - y \leq 13.239\}$

Note that the above-mentioned invariants have an inequality on $x - y$. This is because the loop has the property that the difference between x and y remains constant. Specifically, if x and y start the loop body with values x_1 and y_1, respectively, they will update to $x_2 = x_1 + y_1$ and $y_2 = 2 * y_1$ by the end of the loop body, thus preserving the difference as $x_2 - y_2 = x_1 - y_1$. The property that $x - y$ remains constant depends on the affine transformations of x and y. The fact that our invariant search was guided towards invariants on $x - y$ shows that our neural transformer captures the semantics of affine correctly while also providing differentiability to guide the search for valid invariants.

More precise invariants can be learned through better initialization strategies and by integrating the learned octagon's precision into the learning goal. However, this example highlights the effectiveness of the differentiability of our neural transformers in a practical setting of finding valid octagonal loop invariants.

CHAPTER 6: RELATED WORKS

6.1 LEARNING ABSTRACT TRANSFORMERS

Works like [16, 17] have been proposed to synthesize abstract transformers automatically. These expect the concrete domain and semantics as inputs and use symbolic methods to find exact abstract transformers from a set of functions specified by a Domain Specific Language (DSL). However, sound and precise abstract transformers for many operations like affine assignments in the octagon domain cannot be straightforwardly represented by simple DSL functions. In contrast, our `NeurAbs` framework exploits the ability of neural networks to approximate complex functions ([12, 13]), and proposes a data-driven approach to learn neural transformers with varying soundness and precision.

[18] also uses a data-driven, counter-example guided learning method to learn static analyzers, but the learned static analyzers are symbolic and not differentiable. [19] uses a data-driven approach to learn a neural policy that allows it to remove redundant constraints from abstract states to achieve order of magnitude speedups, but it does not use neural networks to replace the hand-crafted transformers.

6.2 NEURAL SURROGATES

In recent years, significant advancements have been made in developing and applying *neural surrogates for programs* [20, 21], focusing on their potential to speed up program execution [22, 23] and estimate program gradients [24, 25]. Our framework `NeurAbs` lifts this idea from concrete programs to abstract programs. The idea is that the learned neural abstract transformers for each program operation can be composed to obtain a *neural surrogate of the abstract program.* These neural abstract surrogates can enhance the speed and precision of verification processes and facilitate the application of gradient-guided learning techniques for various use cases.

CHAPTER 7: CONCLUSION AND FUTURE WORKS

7.1 CONCLUSION

In this thesis, we introduce the novel concept of *Neural Abstract Transformers*, which are neural networks learned to function as abstract transformers. The `NeurAbs` framework we developed supports both supervised and unsupervised methods for training neural transformers that are both sound and precise. We demonstrated the framework's capability to automatically learn abstract transformers by instantiating it for the Interval and Octagon domains and learning sound and precise transformers for these domains. Additionally, we demonstrated the benefits of the differentiability of neural transformers by employing them to generate octagonal invariants for a loop program. Using neural abstract transformers allows us to frame the tasks of invariant generation as a learning problem, and we then utilize gradient-guided learning to learn the octagonal invariants.

7.2 FUTURE WORKS

1. We demonstrated the benefits of the differentiability of the neural transformers by using them to generate invariants. However, these neural transformers can also be used as a fast and sometimes even more precise replacement for the hand-crafted transformers in analysis tasks.

2. The tensor representation of octagons used by the `NeurAbs` framework (Section 4.2.1) is not efficient as it uses $4*n^2$ values to represent an octagon with n variables. This will not work well for programs with many (say 50-60) variables. More efficient representations, like graphs to represent the inequalities (and using GNNs as neural transformers), can be explored to make this efficient.

3. We instantiated the `NeurAbs` framework for the Interval and the Octagon domain. Extending this framework to include other domains, such as Zonotopes and Polyhedra, will require some innovative approaches and is left for future work.

4. We demonstrated invariant generation as one example of *differentiable abstract interpretation*, which is enabled by learning neural transformers. Other uses of differentiable abstract interpretation, like optimal program synthesis, can also be explored.

REFERENCES

[1] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977. [Online]. Available: https://doi.org/10.1145/512950.512973 p. 238–252.

[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The astreé analyzer," in *Programming Languages and Systems*, M. Sagiv, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 21–30.

[3] P. Cousot and R. Cousot, "Abstract interpretation based program testing," in *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference.* Compact disk paper 248 and electronic proceedings http://www.ssgrr.it/en/ssgrr2000/proceedings.htm, L'Aquila, Italy: Scuola Superiore G. Reiss Romoli, July 31 – August 6 2000.

[4] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290354

[5] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 3–18.

[6] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, p. 253–291, mar 1997. [Online]. Available: https://doi.org/10.1145/244795.244800

[7] A. Mine, "The octagon abstract domain," in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 310–319.

[8] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '78. New York, NY, USA: Association for Computing Machinery, 1978. [Online]. Available: https://doi.org/10.1145/512760.512770 p. 84–96.

[9] P. Cousot and R. Cousot, "Static determination of dynamic properties of generalized type unions," in *Proceedings of an ACM Conference on Language Design for Reliable Software.* New York, NY, USA: Association for Computing Machinery, 1977. [Online]. Available: https://doi.org/10.1145/800022.808314 p. 77–94.

[10] G. Singh, M. Püschel, and M. T. Vechev, "Fast polyhedra abstract domain," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, G. Castagna and A. D. Gordon, Eds. ACM, 2017, pp. 46–59.

[11] G. Singh, M. Püschel, and M. T. Vechev, "Making numerical program analysis fast," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. M. Blackburn, Eds. ACM, 2015, pp. 303–313.

[12] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, p. 359–366, jul 1989.

[13] G. V. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, 1989. [Online]. Available: https://api.semanticscholar.org/CorpusID:3958369

[14] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, Mar 2006. [Online]. Available: https://doi.org/10.1007/s10990-006-8609-1

[15] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, Feb. 2009, vol. 185, ch. 26, pp. 825–885.

[16] P. K. Kalita, S. K. Muduli, L. D'Antoni, T. Reps, and S. Roy, "Synthesizing abstract transformers," vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563334

[17] J. Lim and T. Reps, "Tsl: A system for generating abstract interpreters and its application to machine-code analysis," vol. 35, no. 1, apr 2013. [Online]. Available: https://doi.org/10.1145/2450136.2450139

[18] P. Bielik, V. Raychev, and M. Vechev, "Learning a static analyzer from data," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 233–253.

[19] J. He, G. Singh, M. Püschel, and M. Vechev, "Learning fast and precise numerical analysis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3386016 p. 1112–1127.

[20] A. Renda, Y. Ding, and M. Carbin, "Programming with neural surrogates of programs," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3486607.3486748 p. 18–38.

[21] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 449–460.

[22] C. Mendis, C. Yang, Y. Pu, D. Amarasinghe, and M. Carbin, "Compiler auto-vectorization with imitation learning," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/d1d5923fc822531bbfd9d87d4760914b-Paper.pdf

[23] A. Munk, A. Åšcibior, A. G. Baydin, A. Stewart, G. Fernlund, A. Poursartip, and F. Wood, "Deep probabilistic surrogate networks for universal simulator approximation," in *International Conference on Probabilistic Programming (PROBPROG 2020), Cambridge, MA, United States*, 2020. [Online]. Available: https://probprog.cc/

[24] A. Renda, Y. Chen, C. Mendis, and M. Carbin, "Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2020. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MICRO50266.2020.00045 pp. 442–455.

[25] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," 05 2019, pp. 803–817.

APPENDIX A: PROOFS

Theorem A.1. If y_i is a sound output of an abstract transformer $\hat{op} : \mathcal{A} \to \mathcal{A}$ on some input x_i and $y_i \sqsubseteq_A y_i'$, then y_i' is also a sound output of $\hat{op}$ on x_i.

Proof. To prove that y_i' is also a sound output of $\hat{op}$ on x_i, it is sufficient to prove that $\alpha(op(\gamma(x_i))) \sqsubseteq_A y_i'$ (by Eq. 2.3).

$$\alpha(op(\gamma(x_i))) \sqsubseteq_A y_i \quad \text{(Definition of } y_i \text{ being sound output of } \hat{op} \text{ by Eq. 2.3)} \tag{A.1}$$

$$y_i \sqsubseteq_A y_i' \quad \text{(Given)} \tag{A.2}$$

$$\alpha(op(\gamma(x_i))) \sqsubseteq_A y_i' \quad \text{(From (A.1) \& (A2))} \tag{A.3}$$

QED.

Theorem A.2. Given two octagons oct_1 and oct_2, if there is no inequality i that is stricter in oct_2, then $oct_1 \subseteq oct_2$.

Proof. To prove this, we prove that if a concrete point c belongs to oct_1, it also belongs to oct_2.

1. If c belongs to $oct_1 = [w, e]$, that means that $v_i(c) \leq w_i$ for all i in the set of inequalities present in oct_1, given by $ineq_1 = \{i \mid e_i = 1\}$.

2. As there are no inequalities that are stricter in $oct_2 = [w', e']$, it follows from the definition of strictness (1) that the of inequalities present in oct_2, given by $ineq_2 = \{i \mid e_i' \geq 0.5\}$ is a subset of $ineq_1$.

3. So, for all inequalities in oct_2, $v_i(c) \leq w_i$ holds (from (1) and the fact that $ineq_2 \subseteq ineq_1$).

4. As no inequality is stricter in oct_2, it also means that $w_i \leq w_i'$ for all $i \in ineq_2$.

5. From (3) and (4), we can conclude that $\forall i \in ineq_2, v_i(c) \leq w_i'$. This proves that c also belongs to oct_2.

QED.