Neural Abstract Interpretation

Shaurya Gomber & Gagandeep Singh
Department of Computer Science
University of Illinois Urbana-Champaign
Champaign, IL 61820, USA
{sgomber2,ggnds}@illinois.edu

Abstract

Abstract interpretation is a widely used method for the formal analysis and verification of programs and neural networks. However, designing efficient abstract transformers for widely-used expressive domains such as Octagon and Polyhedra is challenging as one needs to carefully balance the fundamental trade-off between the cost, soundness, and precision of the transformer for downstream tasks. Further, scalable implementations involve intricate performance optimizations like Octagon and Polyhedra decomposition. Given the inherent complexity of abstract transformers and the proven capability of neural networks to effectively approximate complex functions, we envision and propose the concept of *Neural Abstract Transformers*: neural networks that serve as abstract transformers. The proposed Neural Abstract Interpretation (`NAI`) framework provides supervised and unsupervised methods to learn efficient neural transformers *automatically*, which reduces development costs. We instantiate the `NAI` framework for two widely used numerical domains: Interval and Octagon. Evaluations on these domains demonstrate the effectiveness of the `NAI` framework to learn sound and precise neural transformers. An added advantage of our technique is that neural transformers are differentiable, unlike hand-crafted alternatives. As an example, we showcase how this differentiability allows framing invariant generation as a learning problem, enabling neural transformers to generate valid octagonal invariants for a program.

1 Introduction

Abstract Interpretation (Cousot & Cousot, 1977a) is a popular technique for formally analyzing the properties of programs (Cousot et al., 2005; Cousot & Cousot, 2000), neural networks (Singh et al., 2019; Gehr et al., 2018), and complex real-world systems (Dams et al., 1997). Abstract Interpretation works by soundly approximating the *concrete* semantics of the system within a "suitably finite" domain, called the *abstract domain*. The "finiteness" of the abstract domain allows us to reason about all possible executions of the systems efficiently. Analyzing programs in the abstract domain requires that the set of states in the concrete domain are mapped to their *abstraction* in the abstract domain. Similarly, it requires functions that transform one abstract state into another, corresponding to operations in the concrete domain. These functions, known as *Abstract Transformers*, must soundly approximate their concrete counterparts to ensure the correctness of the analysis. The choice of the abstract domain used for analysis is based on the specific properties to be proven. For example, in the analysis of numerical programs, abstract domains such as Octagons (Mine, 2001) and Polyhedra (Cousot & Halbwachs, 1978) are beneficial for verifying intricate program properties because they account for inter-variable dependencies, unlike the Interval domain (Cousot & Cousot, 1977b), which solely represents variable bounds.

Designing efficient abstract transformers for expressive relational domains is challenging, requiring a careful balance of the fundamental trade-off between soundness, precision, and efficiency of the transformer. Sound and precise abstract transformers for operations like join in the Polyhedra and Octagon domains are computationally expensive. Polyhedra join requires computing the convex hull, an exponential-time operation in the number of variables and constraints (Singh et al., 2017), while Octagon join involves closure computation with a cubic-time complexity (Mine, 2001). Using these implementations makes the program analysis task expensive and, thus, less scalable. Various

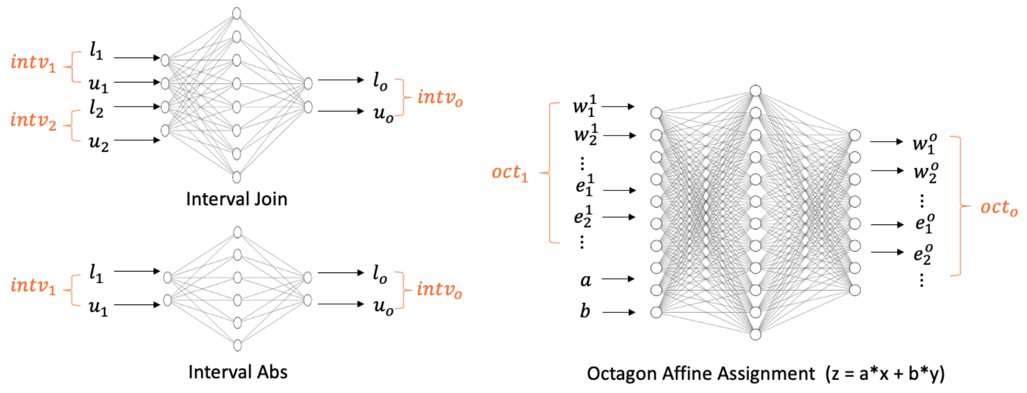

Figure 1: The NAI framework provides supervised and unsupervised learning methods to train neural networks to serve as abstract transformers for abstract interpretation

performance optimizations like Octagon and Polyhedra decomposition (Singh et al., 2015; 2017) have been proposed to make these transformers scalable. However, implementing these optimizations is highly complex, demanding significant manual effort and increasing the risk of soundness bugs. For certain operators, such as affine assignment in the Octagon domain, designing efficient abstract transformers without losing precision is challenging. As a result, practical implementations often favor efficiency over precision, ultimately reducing the overall precision of program analysis.

To tackle the challenges of designing efficient abstract transformers, we propose a data-driven learning approach to automate their generation. Inspired by the ability of neural networks to learn complex functions (Hornik et al., 1989; Cybenko, 1989), we introduce Neural Abstract Interpretation (NAI), a framework that learns **Neural Abstract Transformers** – neural networks that serve as the abstract transformers. The NAI framework offers three key advantages: (1) *Automatic Transformer Generation* – it leverages supervised and unsupervised learning to train neural abstract transformers, reducing manual effort while providing mechanisms to balance the soundness-precision trade-off; (2) *Fast and Precise Transformers* – neural transformers can surpass hand-crafted ones in speed and, in some cases, precision, enhancing scalability while defaulting to traditional methods when they are unsound; and (3) *Differentiability* – unlike conventional transformers, neural abstract transformers are differentiable, enabling gradient-based learning for tasks like invariant generation and paving the way for *differentiable abstract interpretation*. Our contributions are as follows:

1. We pose the problem of learning sound and precise abstract transformers as an optimization problem (Sec. 2) and point out why the general optimization problem is hard to solve. We introduce a general framework NAI (Sec. 2) that proposes supervised (Sec. 2.1) and unsupervised (Sec. 2.2) 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.
2. We instantiate our NAI framework for the Interval and the Octagon domain (details in Appendix C). We demonstrate the effectiveness of our supervised and unsupervised learning methods by learning sound and precise neural transformers for operators in the Interval and Octagon domains (Sec. 3.1). 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. 3.2).

2 NAI Framework

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 $\mathcal{A}$. $\hat{op}$ is *sound* if it over-approximates the output of *op*, i.e. $\forall a \in \mathcal{A}.\ op(\gamma(a)) \sqsubseteq_C \gamma(\hat{op}(a))$ where $\gamma : \mathcal{A} \to \mathcal{C}$ is the concretization function (more details in Appendix A). 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 $\hat{op}$. Precision is important as any abstract transformer that returns $\top$ is technically sound but is not very useful for practical settings. We ideally need sound transformers that are as precise as possible.

Now, 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 & Cousot (1977a), we represent the "most-precise abstract transformer" for op as $\hat{op}^{\#}$. 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:

$$\hat{op} = \arg\min_{f \in \mathcal{F}} \sum_{a \in \mathcal{A}} \mathcal{L}_P(\hat{op}^{\#}(a),\ f(a)) \ \text{ s.t.} \sum_{a \in \mathcal{A}} \mathcal{L}_S(op,\ a,\ f(a)) = 0 \tag{1}$$

where:

1. $\mathcal{L}_S(op, a, f(a))$ measures the soundness of f. It returns 0 if $f(a)$ soundly approximates op on a, i.e., $op(\gamma(a)) \sqsubseteq_C \gamma(f(a))$, where γ is the concretization function, and 1 if $f(a)$ is unsound. Thus, $\sum_{a \in \mathcal{A}} \mathcal{L}_S(a, f(a)) = 0$ ensures f is sound for all $a \in \mathcal{A}$.
2. $\mathcal{L}_P(a_1, a_2)$ measures precision and can be any metric to measure how "big" is a_2 compared to a_1. For intervals, this can be the difference in interval sizes. Minimizing $\sum_{a \in \mathcal{A}} \mathcal{L}_P(\hat{op}^{\#}(a), f(a))$ finds an abstract transformer closest in precision to the most precise one, $\hat{op}^{\#}$. If $\hat{op}^{\#} \in \mathcal{F}$, then it is the optimal solution.

Inherent complexity. While this approach to learning abstract transformers is correct, solving the optimization problem is challenging. First, the abstract domain is often infinite, making it difficult to satisfy both soundness and precision for all elements. Second, Cousot & Cousot (1977a) provides only a specification of $\hat{op}^{\#}$ but no concrete method to compute it, thus complicating the computation of $\mathcal{L}_P$. Finally, computing $\mathcal{L}_S$ involves finding counterexamples to soundness via SMT, which is computationally expensive and non-differentiable, making gradient-based learning infeasible.

To address the complexity of the learning problem, we leverage neural networks' ability to approximate complex functions (Hornik et al., 1989; Cybenko, 1989) and reframe the design of sound and precise abstract transformers as a more tractable learning task. In the following sections, we introduce supervised and unsupervised relaxations to achieve this.

2.1 Supervised Learning of Neural Transformers

In this section, we present a supervised learning approach for training neural abstract transformers, specifically useful for operators that have hand-crafted sound and precise abstract transformers.

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:

$$\min_{\theta} \mathbb{E}_{(X_i, y_i) \sim D} \left[\alpha * \mathcal{L}'_S(y_i,\ \hat{op}^*(X_i;\theta)) + \beta * \mathcal{L}'_P(y_i,\ \hat{op}^*(X_i;\theta))\right] \tag{2}$$

which is based on the following components:

1. Soundness Loss $\mathcal{L}'_S$: Given ground truth outputs y_i of the abstract transformer, soundness is ensured by enforcing that the output of the model $\hat{op}^*(X_i;\theta)$ over-approximates y_i, i.e., $y_i \sqsubseteq_A \hat{op}^*(X_i;\theta)$ (by Theorem 1). We define $\mathcal{L}'_S$ such that $\mathcal{L}'_S(a_1, a_2) = 0$ implies $a_1 \sqsubseteq_A a_2$, which holds if $\gamma(a_1) \sqsubseteq_C \gamma(a_2)$. If $\mathcal{L}'_S(a_1, a_2) \neq 0$, it provides a differentiable approximation of $\gamma(a_1) \setminus \gamma(a_2)$, guiding the model to minimize $\mathcal{L}'_S$ and ensure over-approximation. For example, in the Interval domain, if the ground truth is $[0, 5]$ and the model predicts $[0, 4]$, the soundness loss is $5 - 4 = 1$, encouraging the model to expand the upper bound. Minimizing $\mathcal{L}'_S(y_i, \hat{op}^*(X_i;\theta))$ ensures the neural transformer's output over-approximates the ground truth, enforcing soundness. Unlike $\mathcal{L}_S(op, a, f(a))$ in Eq. 1, which requires encoding op's semantics, $\mathcal{L}'_S$ is differentiable and avoids explicit dependence on op, leveraging the supervised setting with known ground truths.

2. Precision Loss $\mathcal{L}'_P$: Given a measure $\mathcal{M}(a)$ of the size of $\gamma(a)$ for $a \in \mathcal{A}$, $\mathcal{L}'_P(a_1, a_2)$ quantifies how much larger $\mathcal{M}(a_2)$ is compared to $\mathcal{M}(a_1)$. While ensuring $\gamma(a_1) \sqsubseteq_C \gamma(a_2)$ guarantees soundness, an overly imprecise transformer (e.g., always outputting $\top_{\mathcal{A}}$) is unhelpful for analysis. To enforce precision, $\mathcal{L}'_P(a_1, a_2) \geq 0$ provides a differentiable estimate of over-approximation given by $\max(\mathcal{M}(a_2) - \mathcal{M}(a_1), 0)$. In the Interval domain, the precision loss can be the interval size difference, with $\mathcal{M}([l, u]) = \max(u - l, 0)$. Minimizing $\mathcal{L}'_P(y_i, \hat{op}^*(X_i; \theta))$ ensures the neural transformer's output stays closer to the ground truth in precision. Unlike $\mathcal{L}_P(\hat{op}^{\#}(a), f(a))$ in Eq. 1, this formulation avoids computing $\hat{op}^{\#}$ for precision assessment, relying instead on ground truth outputs to measure and minimize imprecision.

3. Soundness & Precision Weights: α, β control the required degree of soundness and precision, allowing adjustment based on the specific downstream task for which the neural transformer is used.

Appendix C demonstrates that $\mathcal{L}'_S(a_1, a_2)$ and $\mathcal{L}'_P(a_1, a_2)$ can be computed efficiently and differentiably for operations in abstract domains like Intervals and Octagons.

2.2 Unsupervised Learning of Neural Transformers

We now propose an unsupervised learning algorithm for training neural abstract transformers, which is useful when precise hand-crafted transformers are difficult to implement. Unlike the supervised approach, it does not require ground truth outputs but instead defines soundness and precision losses based on the semantics of the concrete operator *op*.

Learning Problem. Given a dataset $D = \{X_i\}$ representing a 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:

$$\min_{\theta} \mathbb{E}_{X_i \sim D} \left[\alpha * \mathcal{L}''_S(op,\ X_i,\ \hat{op}^*(X_i; \theta)) + \beta * \mathcal{L}''_P(\hat{op}^*(X_i; \theta))\right] \tag{3}$$

which is based on the following components:

1. Soundness Loss $\mathcal{L}''_S$: The key challenge is the absence of ground truth outputs for computing soundness loss. An abstract state a_2 is sound for an operation *op* on a_1 if it over-approximates the effect of *op*, formally $op(\gamma(a_1)) \sqsubseteq_C \gamma(a_2)$. Thus, the soundness loss $\mathcal{L}''_S(op, a_1, a_2)$ should be 0 if this condition holds and otherwise approximate the size of $op(\gamma(a_1)) \setminus \gamma(a_2)$. However, checking this typically requires SMT solvers, which are non-differentiable and unsuitable for guiding learning. To address this, we first define the *distance* $\mathcal{D}(c, a)$ between a concrete point c and an abstract state a, measuring how far c is from being included in $\gamma(a)$. For instance, in the Interval domain, if $a = [2, 4]$ and $c = 7$, then $\mathcal{D}(c, a) = 7 - 4 = 3$, with $\mathcal{D}(c, a) = 0$ when $c \in \gamma(a)$. Now, we define the *maximum violating concrete point (MVCP)* for (op, a_1, a_2) as:

Definition 1. *For a tuple (op, a_1, a_2), where op is a concrete operator and $a_1, a_2 \in \mathcal{A}$, a maximum violating concrete point (MVCP) is the state $c_m \in \mathcal{C}$ that belongs to $op(\gamma(a_1))$ but is missing from $\gamma(a_2)$. It is the farthest such point, measured by $\mathcal{D}(c, a_2)$. If $op(\gamma(a_1)) \subseteq \gamma(a_2)$, meaning a_2 is a sound abstraction, no MVCP exists.*

$$MVCP(op, a_1, a_2) = \underset{c \in op(\gamma(a_1)) \setminus \gamma(a_2)}{\text{argmax}} \mathcal{D}(c, a_2) \tag{4}$$

Using MVCP, the soundness loss is defined as:

$$\mathcal{L}''_S(op, a_1, a_2) = \mathcal{D}(MVCP(op, a_1, a_2), a_2) \tag{5}$$

Minimizing $\mathcal{L}''_S(X_i, \hat{op}^*(X_i; \theta))$ ensures learning a sound transformer by iteratively reducing the distance between $\hat{op}^*(X_i; \theta)$ and the MVCP. Fig 2 (left) illustrates this process: the MVCP (green dot) is the farthest missing point from the concretization (red region). Minimizing the distance d^* guides the model to include MVCPs, ensuring soundness. For example, in the interval domain, while learning $\hat{abs}^*$ for the *abs* operator with $a_1 = [-10, 15]$, the model output $a_2 = [0, 12]$ results in an unsound abstraction since $abs(\gamma(a_1)) = [0, 15]$ should be included. The MVCP is $c_m = 15$, yielding $\mathcal{L}''_S(c_m, a_2) = 15 - 12 = 3$, guiding the model toward a sound transformer.

2. Precision Loss $\mathcal{L}''_P$: Consider a measure $\mathcal{M}(a)$ representing the size of $\gamma(a)$ for an element $a \in \mathcal{A}$. In the unsupervised setting, where ground truth outputs are unavailable, precision is enforced

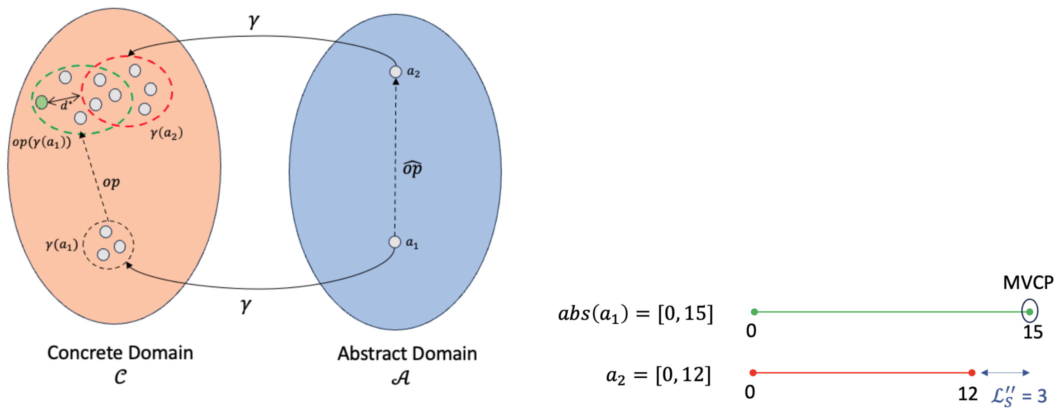

Figure 2: Illustration of the Maximum Violating Concrete Point (MVCP) guiding the learning of neural transformer. In the left figure, a_2 is unsound as $op(\gamma(a_1)) \not\sqsubseteq_C \gamma(a_2)$, with the MVCP (green dot) lying outside $\gamma(a_2)$. Minimizing the loss ensures $\gamma(a_2)$ includes the MVCP. In the right figure, for abs, if $a_1 = [-10, 15]$, the output must contain $[0, 15]$ for soundness. Since a_2 is unsound, the MVCP $c_m = 15$ defines the soundness loss.

by directly minimizing the size $\mathcal{M}(a)$ of the model's output. Specifically, $\mathcal{L}''_P(a)(\geq 0)$ provides a differentiable approximation of $\mathcal{M}(a)$. For instance, in the Interval domain, the size of $[l, u]$ is given by $\mathcal{M}([l, u]) = \max(u - l, 0)$, which can be minimized to improve precision. Thus, minimizing $\mathcal{L}''_P(\hat{op}^*(X_i; \theta))$ guides the neural transformers to produce more precise outputs.

3. Soundness & Precision Weights: α, β control the required degree of soundness and precision, allowing adjustment based on the specific downstream task for which the neural transformer is used.

Appendix C demonstrates that 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. Beyond enabling efficient and differentiable transformers, unsupervised learning offers the added advantage of eliminating the need for hand-crafted transformers. While supervised learning still relies on existing precise transformers, unsupervised learning generates them automatically using only domain functions $\mathcal{D}(c, a)$, $\mathcal{L}''_P(a)$, and the semantics of $MVCP(op, a_1, a_2)$. This is particularly beneficial for complex operations like affine assignment in the Octagon domain, where efficient and precise hand-crafted transformers do not exist but can now be learned as neural transformers.

2.3 Soundness Precision Trade-off

In both supervised and unsupervised learning, the loss has two components: soundness and precision. Optimizing solely for soundness can produce overly large outputs, while precision loss enforces tighter approximations, creating a trade-off. Soundness loss expands outputs to cover the sound region, whereas precision loss shrinks them to maintain precision, making it a challenging multi-objective optimization problem. By tuning soundness-precision weights (α and β), neural transformers can be learned with varying degrees of soundness and precision based on the specific downstream task. In traditional verification tasks, where soundness is crucial, unsound cases can default to hand-crafted transformers, as soundness checks are relatively simple.

3 Evaluation

3.1 Soundness and Precision of Neural Abstract Transformers

In this section, we evaluate the soundness and precision of neural transformers learned via supervised and unsupervised approaches within the `NAI` framework. We train transformers for the abs and $join$ operators in the Interval domain (Sec A.3.1) and the $join$ and *affine assignment* operators in the Octagon domain (Sec A.3.2).

Weights (α, β)	Interval Abs		Interval Join	
(Soundness, Precision)	Soundness (%)	Size Difference	Soundness (%)	Size Difference
(-, -)	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

Table 1: Neural Interval Transformers for Abs and Join trained using the supervised approach

Weights (α, β)	Octagon Join		
(Soundness, Precision)	Soundness (%)	Edge Count Difference	Constants Sum Difference
(-, -)	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 2: Neural Octagon Transformer for Join trained using the supervised approach

Datasets. For interval `abs` and `join`, as well as octagon `join`, training and testing data are generated by randomly sampling input intervals and octagons and computing ground truth outputs using hand-crafted transformers. For octagon *affine assignment*, where no precise hand-crafted alternative exists, training and testing data are generated by only sampling random input octagons and affine assignments, as ground truth is not required in the unsupervised setting.

Evaluation Metrics. *Soundness (%)* denotes the percentage of sound outputs generated by the learned neural transformers on a test set (of size 10,000 for intervals and 1,000 for octagons). Since soundness ensures correctness but not precision, different metrics are used to assess precision for interval and octagonal transformers. For interval transformers, where ground truth is available, we use *Size Difference*, the average difference between the sizes of model output intervals and ground truth intervals, considering only the sound cases. A smaller size difference indicates a more precise model. For octagon transformers, precision is harder to define due to the lack of a straightforward notion of octagon size. Thus, we use the following metrics:

- **Supervised setting**: Since ground truth octagons are available, we compare the learned transformer's output to the true octagon. We measure *Edge Count Difference*, the average difference in the number of inequalities between the learned and ground truth octagons, and *Constants Sum Difference*, the average difference in the sum of inequality constants. Larger differences in either metric indicate a looser, less precise abstraction, whereas smaller values suggest that the output is closer to ground truth in precision.
- **Unsupervised setting**: Without ground truth octagons, we assess precision based on the structure of the learned output. We use *Edge Count*, the average number of inequalities in the transformer's output, and *Constants Sum*, the sum of inequality constants. Fewer inequalities and a higher sum indicate a less precise abstraction.

Supervised Learning. Using the loss functions from Sections C.1.2 and C.2.2, we train neural transformers for both the Interval and Octagon domains. For the Interval domain, we train transformers for abs and join operations on 5,000 samples each, while for the Octagon domain, we train a join transformer on 10,000 samples. Tables 1 and 2 present the results, where the first column lists the soundness and precision weights (α, β), and the first row reports the performance of a randomly

Weights (α, β)	Interval Abs		Interval Join	
(Soundness, Precision)	Soundness (%)	Size Difference	Soundness (%)	Size Difference
(-, -)	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

Table 3: Neural Interval Transformers for Abs and Join trained using the unsupervised approach

Weights (α, β)	Octagon Join		
(Soundness, Precision)	Soundness (%)	Edge Count	Constants Sum
(-, -)	0.0	-	-
(10, 1000)	1.5	8	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 4: Neural Octagon Transformer for affine assignment trained using the unsupervised approach

initialized network. In both cases, random networks fail to generate sound outputs, whereas our supervised approach successfully learns sound and precise transformers. Increasing the soundness weight α improves soundness but reduces precision, as evidenced by the increasing difference in the number of inequalities and the sum of constants in the octagons generated by the learned transformers compared to the ground truth, and by the increasing size difference in the case of intervals. This demonstrates the framework's ability to balance the soundness-precision trade-off and learn neural transformers to varying levels of precision and generalization.

Unsupervised Learning. We use the loss methods described in Sections C.1.3 and C.2.3 to train neural transformers for the interval and octagon domains using 1000 samples each. The dataset size is limited due to the computational cost of computing MVCPs at each iteration, which involves expensive SMT solver calls. Despite this, Tables 3 and 4 confirm that our approach successfully learns sound and precise transformers. For interval transformers, we train for the abs and join operations. Due to the lack of ground truth in the unsupervised setting, models tend to favor sound but imprecise solutions. To counter this, we assign a higher precision weight (10) to achieve results comparable to supervised learning. During training, precision is enforced using the sizes of the output intervals, whereas evaluation compares the sizes of the learned intervals to the ground truth intervals. Similarly, for octagon affine, we use a higher precision weight (1000), and precision is enforced during training using the size of output octagons. Increasing the soundness weight α improves soundness but reduces precision, as seen in fewer inequalities in the octagons generated by learned transformers (and increasing size difference in case of intervals). This again highlights the ability of our framework to balance the soundness-precision trade-off.

3.2 Differentiable Learning of Loop Invariants

This section highlights the advantage of our differentiable neural abstract transformers in learning loop invariants. We frame the task of discovering valid inductive octagonal invariants for a loop program $\mathcal{P}$ as a learning problem. Consider a loop program $\mathcal{P} = \texttt{while}(\beta)\ \texttt{do}\ \mathcal{C}\ \texttt{od}$, where the effective abstract transformer of $\mathcal{C}$ is defined as $\hat{\mathcal{C}} = \hat{op}_n \circ \hat{op}_{n-1} \cdots \circ \hat{op}_1$, with $\hat{op}_i$ representing the abstract transformer for the i^{th} statement in $\mathcal{C}$. If O_{init} approximates the initial states of $\mathcal{P}$, an octagon O_{inv} is a valid invariant if:

$$(O_{init} \subseteq O_{inv}) \ \wedge \ (\hat{\mathcal{C}}(\hat{conj}(O_{inv},\ \beta)) \subseteq O_{inv}) \tag{6}$$

where $\hat{conj}$ is the abstract transformer approximating the conjunction of an octagon with β. Using our NAI framework, we construct neural approximations $\hat{op}_i^*$ for each transformer $\hat{op}_i$, yielding the *effective neural transformer for the loop body* $\hat{\mathcal{C}}^*$ as $\hat{\mathcal{C}}^* = \hat{op}_n^* \circ \hat{op}_{n-1}^* \cdots \circ \hat{op}_1^*$. This enables searching for O_{inv} by minimizing:

$$\mathcal{L}'_S(O_{init},\ o) + \mathcal{L}'_S(\hat{\mathcal{C}}^*(\hat{conj}^*(o, \beta)),\ o) \tag{7}$$

where $\mathcal{L}'_S(o_1, o_2)$ (Sec C.2.2) enforces $o_1 \subseteq o_2$. Starting from random octagons, gradient descent can be used to minimize Eq. 7 and generate candidate invariants, which can then be verified against Eq. 6 via an SMT solver. For example, consider the loop in Fig. 3. The initial state $x = 100, y = 150$ is represented by the octagon $O_{init} = \{x \geq 100, -x \leq -100, y \geq 150, -y \leq -150\}$. Training a neural transformer for affine assignments in the octagon domain and using Eq. 7, we synthesize non-trivial invariants such as:

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

Figure 3: Example loop program

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

These invariants capture the loop's structural property that $x - y$ remains constant, as updates follow $x_2 = x_1 + y_1$ and $y_2 = 2y_1$, preserving $x_2 - y_2 = x_1 - y_1$. The emergence of these constraints demonstrates that our neural transformers effectively capture loop semantics while maintaining differentiability to guide invariant synthesis. More precise invariants can be learned with improved initialization and precision-aware learning. However, this example highlights the effectiveness of differentiable neural transformers in a practical setting of finding valid octagonal loop invariants.

4 Related Works

Works like Kalita et al. (2022); Lim & Reps (2013) use symbolic methods to synthesize abstract transformers from a predefined DSL, requiring explicit concrete semantics. However, many operations, such as affine assignments in the Octagon domain, cannot be easily represented with simple DSL functions. In contrast, our NAI framework leverages neural networks' ability to approximate complex functions (Hornik et al., 1989; Cybenko, 1989), adopting a data-driven approach to learn neural transformers with varying soundness and precision. While Bielik et al. (2017) applies counterexample-guided learning to static analysis, their analyzers remain symbolic and non-differentiable. Similarly, He et al. (2020) learns a neural policy to optimize abstract states but does not replace hand-crafted transformers with neural networks.

Recent advancements in *neural surrogates* (Renda et al., 2021; Esmaeilzadeh et al., 2012) have focused on accelerating program execution (Mendis et al., 2019; Munk et al., 2020) and estimating program gradients (Renda et al., 2020; She et al., 2019). Our NAI framework lifts this idea from concrete to abstract programs by composing learned neural abstract transformers for each program operation to obtain a *neural surrogate of the abstract program.* These surrogates can improve verification speed and precision while enabling gradient-guided learning for various applications.

5 Conclusion and Future Works

This paper introduced *Neural Abstract Transformers*, neural networks trained to serve as abstract transformers within the NAI framework, which supports both supervised and unsupervised learning. We demonstrated its ability to automatically learn sound and precise transformers for the Interval and Octagon domains and leveraged their differentiability for invariant generation via gradient-guided learning. Beyond invariant generation, neural transformers can serve as fast and, in some cases, more precise alternatives to hand-crafted transformers in analysis tasks. However, the current tensor representation of octagons in NAI (Section C.2.1) is inefficient, requiring $4n^2$ values for n variables, making it impractical for large programs. Exploring graph-based representations, such as GNNs, to encode inequalities could improve efficiency. While we instantiated NAI for the Interval and

Octagon domains, extending it to more expressive domains such as Zonotopes and Polyhedra is a promising direction. Additionally, this work demonstrates the potential of *differentiable abstract interpretation*, enabling new applications in optimal program synthesis and other analysis tasks.

References

Pavol Bielik, Veselin Raychev, and Martin Vechev. Learning a static analyzer from data. In Rupak Majumdar and Viktor Kunčak (eds.), *Computer Aided Verification*, pp. 233–253, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63387-9.

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, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli. ISBN 88-85280-52-8.

Patrick Cousot and Radhia 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*, POPL '77, pp. 238–252, New York, NY, USA, 1977a. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL `https://doi.org/10.1145/512950.512973`.

Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pp. 77–94, New York, NY, USA, 1977b. Association for Computing Machinery. ISBN 9781450373807. doi: 10.1145/800022.808314. URL `https://doi.org/10.1145/800022.808314`.

Patrick Cousot and Nicolas 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*, POPL '78, pp. 84–96, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450373487. doi: 10.1145/512760.512770. URL `https://doi.org/10.1145/512760.512770`.

Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Mooly Sagiv (ed.), *Programming Languages and Systems*, pp. 21–30, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31987-0.

George V. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989. URL `https://api.semanticscholar.org/CorpusID:3958369`.

Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, mar 1997. ISSN 0164-0925. doi: 10.1145/244795.244800. URL `https://doi.org/10.1145/244795.244800`.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, 2012. doi: 10.1109/MICRO.2012.48.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 3–18, 2018. doi: 10.1109/SP.2018.00058.

Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. Learning fast and precise numerical analysis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pp. 1112–1127, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386016. URL `https://doi.org/10.1145/3385412.3386016`.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, jul 1989. ISSN 0893-6080.

Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. Synthesizing abstract transformers. 6(OOPSLA2), oct 2022. doi: 10.1145/3563334. URL `https://doi.org/10.1145/3563334`.

Junghee Lim and Thomas Reps. Tsl: A system for generating abstract interpreters and its application to machine-code analysis. 35(1), apr 2013. ISSN 0164-0925. doi: 10.1145/2450136.2450139. URL `https://doi.org/10.1145/2450136.2450139`.

Charith Mendis, Cambridge Yang, Yewen Pu, Dr.Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper_files/paper/2019/file/d1d5923fc822531bbfd9d87d4760914b-Paper.pdf`.

A. Mine. The octagon abstract domain. In *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 310–319, 2001. doi: 10.1109/WCRE.2001.957836.

Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, Mar 2006. ISSN 1573-0557. doi: 10.1007/s10990-006-8609-1. URL `https://doi.org/10.1007/s10990-006-8609-1`.

Andreas Munk, Adam Åšcibior, AtÄ±lÄ±m GÃ¼neÅŸ Baydin, Andrew Stewart, Goran Fernlund, Anoush Poursartip, and Frank Wood. Deep probabilistic surrogate networks for universal simulator approximation. In *International Conference on Probabilistic Programming (PROBPROG 2020), Cambridge, MA, United States*, 2020. URL `https://probprog.cc/`.

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)*, pp. 442–455, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society. doi: 10.1109/MICRO50266.2020.00045. URL `https://doi.ieeecomputersociety.org/10.1109/MICRO50266.2020.00045`.

Alex Renda, Yi Ding, and Michael 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*, Onward! 2021, pp. 18–38, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391108. doi: 10.1145/3486607.3486748. URL `https://doi.org/10.1145/3486607.3486748`.

Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. pp. 803–817, 05 2019. doi: 10.1109/SP.2019.00052.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In David Grove and Stephen M. Blackburn (eds.), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 303–313. ACM, 2015.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Fast polyhedra abstract domain. In Giuseppe Castagna and Andrew D. Gordon (eds.), *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pp. 46–59. ACM, 2017.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290354. URL `https://doi.org/10.1145/3290354`.

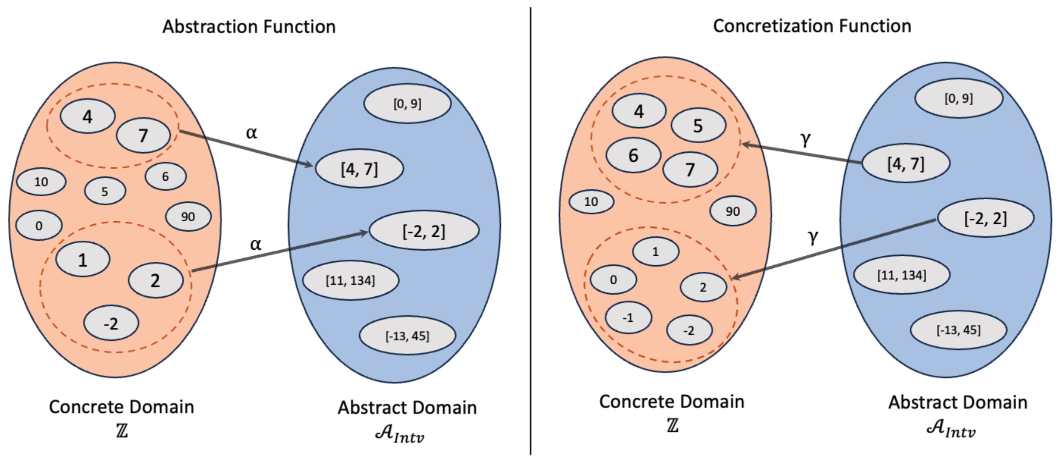

Figure 4: Abstraction and Concretization Functions to abstract integers in the interval domain. It also demonstrates the loss of precision introduced by concretizing back the abstract elements as the set returned after concretizing $[-2, 2]$ has elements that were not there in the original set.

A BACKGROUND ON ABSTRACT INTERPRETATION

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 ensure the *soundness* of abstract interpretation. We then introduce Abstract Transformers, discussing their *soundness* and *precision*—key aspects for understanding the problem we address. The section concludes with an overview of commonly used abstract numerical domains, two of which, Interval and Octagon, are used in our evaluation.

A.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 (Cousot & Cousot, 1977a). 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 4. 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 loss of precision introduced when concretizing the abstract elements.

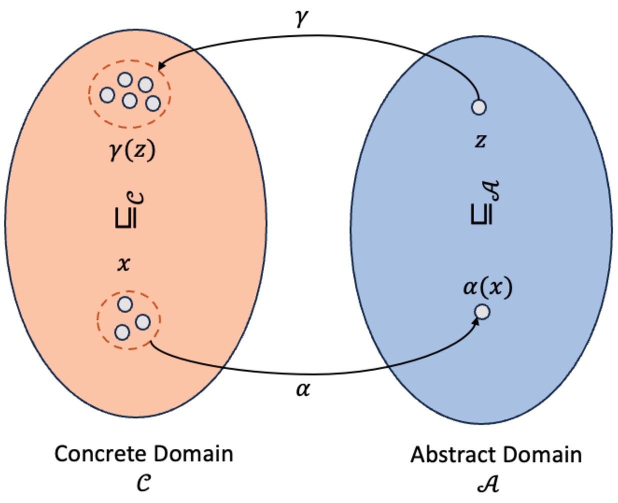

Figure 5: Galois Connection

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. *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:*

$$\forall x \in \mathcal{P}(\mathcal{C}).\ \forall z \in \mathcal{A}.\ \alpha(x) \sqsubseteq_A z \iff x \sqsubseteq_C \gamma(z) \tag{8}$$

Intuitively, this means that α and γ respect the orderings of $\mathcal{P}(\mathcal{C})$ and $\mathcal{A}$ as illustrated in Fig 5. The following directly follows from the above definition (by substituting $z = \alpha(x)$):

$$\forall x \in \mathcal{P}(\mathcal{C}).\ x \sqsubseteq_C \gamma(\alpha(x)) \tag{9}$$

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.

A.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 $\mathcal{P}(\mathcal{C}) \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \mathcal{A}$, the soundness condition for $\hat{op}$ can be mathematically defined as:

$$\forall z \in \mathcal{A}.\ \alpha(op(\gamma(z))) \sqsubseteq_A \hat{op}(z) \tag{10}$$

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: $\gamma(z)$.

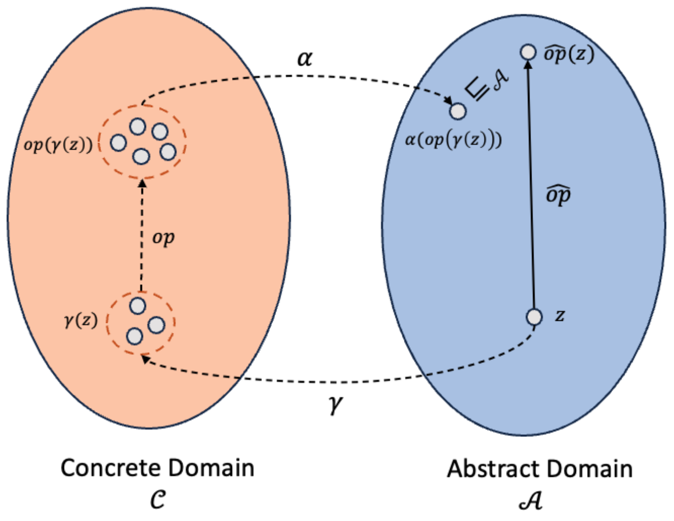

Figure 6: Soundness of Abstract Transformer $\hat{op}$

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 $\alpha(op(\gamma(z)))$, because intuitively, $\alpha(op(\gamma(z)))$ represents the *smallest* abstract element that covers all possible concrete values that can be generated by op. Cousot & Cousot (1977a) 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 8), the soundness condition in Eq. 10 can be re-written only in terms of γ as follows:

$$\forall z \in \mathcal{A}.\ op(\gamma(z)) \sqsubseteq_C \gamma(\hat{op}(z)) \tag{11}$$

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 $\hat{op}$.

A.3 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.

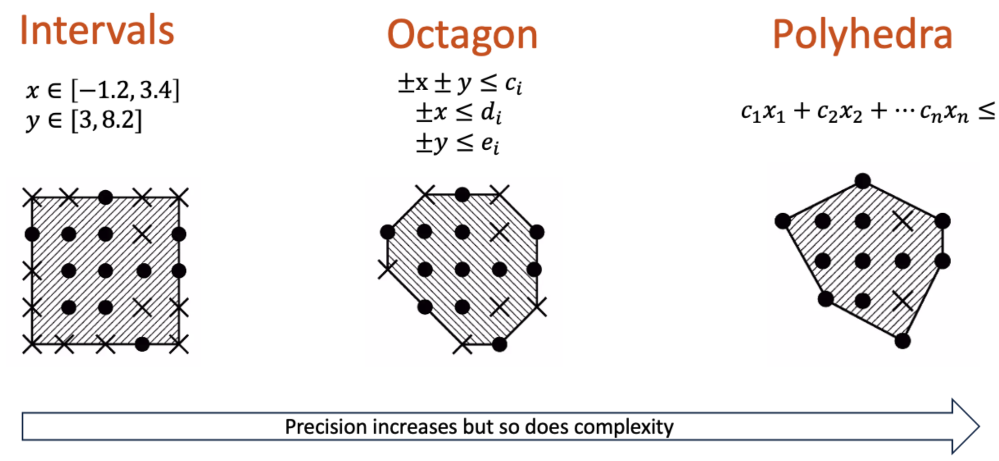

Figure 7: Abstracting the set of black dots using the different abstract domains (Miné (2006)). The crosses indicate the extra points that are added in the abstraction and that lead to imprecision. As we go right, the precision increases but so does the domain complexity.

A.3.1 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 7).

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 $[min(l_1, l_2), max(u_1, u_2)]$.

A.3.2 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. $\pm v_i \pm v_j \le c_{ij}$: Between any 2 variables and the coefficients can only be ± 1.
2. $\pm v_i \le d_i$: 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 $O(n^3)$ (Mine (2001)). Other transformers, like affine assignment in the octagon domain, take an octagon o and return the resultant octagon o' after computing expressions such as $z = a * x + b * y$.

A.3.3 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 Singh et al. (2017).

B Theorems

Theorem 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. 10).

$$\alpha(op(\gamma(x_i))) \sqsubseteq_A y_i \quad \text{(Definition of } y_i \text{ being sound output of } \hat{op} \text{ by Eq. 10)} \tag{12}$$

$$y_i \sqsubseteq_A y_i' \quad \text{(Given)} \tag{13}$$

$$\alpha(op(\gamma(x_i))) \sqsubseteq_A y_i' \quad \text{(From (A.1) \& (A2))} \tag{14}$$

□

Theorem 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.

□

C Instantiation for numerical domains

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

C.1 Interval Domain

In the Interval domain [Sec A.3.1], 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.

C.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 $[l_o, u_o]$.

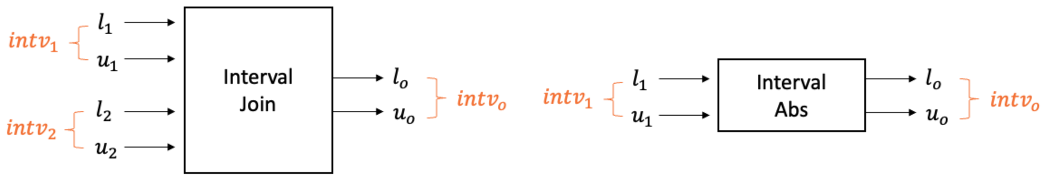

Figure 8: Neural Transformers for Interval Join and Interval Abs

C.1.2 Supervised Learning of Neural Interval Transformers

The general supervised learning approach described in Section 2.1 can be used to learn neural transformer for the Interval domain by using the following instantiations of the loss components present in Eq. 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 $l_2 \le l_1$ and $u_2 \ge u_1$. 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:
$$\mathcal{L}'_S([l_1, u_1],\ [l_2, u_2]) = max(l_2 - l_1, 0) + max(u_1 - u_2, 0) \tag{15}$$
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 $l_2 \le l_1$ and $u_2 \ge u_1$, 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:
$$\begin{aligned}\mathcal{L}'_P([l_1, u_1],\ [l_2, u_2]) &= \max(\mathcal{M}([l_2, u_2]) - \mathcal{M}([l_1, u_1]), 0)\\ &= \max(\max(u_2 - l_2, 0) - \max(u_1 - l_1, 0), 0)\end{aligned} \tag{16}$$
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.

C.1.3 Unsupervised Learning of Neural Interval Transformers

The general unsupervised learning approach described in Section 2.2 can be used to learn neural transformer for the Interval domain by using the following instantiations of the loss components present in Eq. 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:

   $$\mathcal{D}(c, [l, u]) = \begin{cases} l - c & \text{if } c < l \\ c - u & \text{if } u < c \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

   $\mathcal{D}(c, [l, u])$ 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:

   $$\begin{aligned} &\underset{c}{argmax} && \mathcal{D}(c, [l_2, u_2]) \\ &\text{subject to} && \exists x.\ l_1 \le x \le u_1\ \wedge\ c = |x| \end{aligned} \tag{18}$$

   (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:

   $$\begin{aligned} &\underset{c}{argmax} && \mathcal{D}(c, [l_3, u_3]) \\ &\text{subject to} && (l_1 \le c \le u_1)\ \vee\ (l_2 \le c \le u_2) \end{aligned} \tag{19}$$

   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. 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

   $$\mathcal{L}''_P([l, u]) = \mathcal{M}([l, u]) = \max(u - l, 0) \tag{20}$$

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

C.2 Octagon Domain

In the Octagon domain [Sec A.3.2], 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 $\pm v_i \pm v_j \le c_{ij}$ and $\pm v_i \le d_i$.

C.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 `NAI` 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.

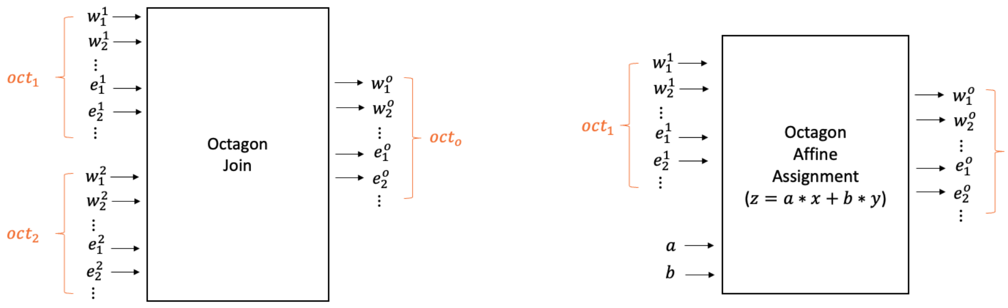

Figure 9: Neural Transformers for Octagon Join and Affine Assignment

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 $\geq$ 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.

C.2.2 Supervised Learning of Neural Octagon Transformers

The general supervised learning approach described in Section 2.1 can be used to learn neural transformer for the Octagon domain by using the following instantiations of the loss components present in Eq. 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 2). To enforce this, we first define the contribution of the i^{th} inequality to soundness loss as follows:

$$l_i = \begin{cases} k_1 * (w_i - w'_i) & \text{if } e'_i \geq 0.5 \text{ and } e_i = 1 \text{ and } w_i > w'_i \\ k_2 * BCELoss(e'_i, e_i) & \text{if } e'_i \geq 0.5 \text{ and } e_i = 0 \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

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:

$$k_1 * BCELoss(e'_i, e_i \mid e_i = 1) + k_2 * \sum_i (max(w'_i - w_i,\ 0)) \tag{22}$$

   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.

C.2.3 Unsupervised Learning of Neural Octagon Transformers

The general unsupervised learning approach described in Section 2.2 can be used to learn neural transformer for the Octagon domain by using the following instantiations of the loss components present in Eq. 3:

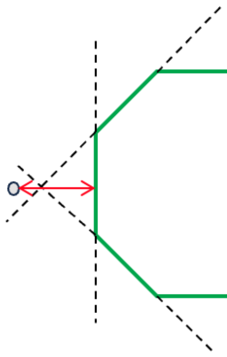

Figure 10: The dotted lines show the inequalities the point does not satisfy. The distance in red shows the $\mathcal{D}(c, oct)$.

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

   $$\delta(c, [w, e]) = \{i \mid e_i = 1 \ \wedge \ v_i(c) > w_i\} \tag{23}$$

   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.

   $$\mathcal{D}(c, [w, e]) = \underset{i \in \delta(c,[w,e])}{max} \ v_i(c) - w_i \tag{24}$$

   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:

   $$\begin{aligned} &\underset{c}{argmax} && \mathcal{D}(c, oct_2) \\ &\text{subject to} && \exists v_1, v_2.\ encode(oct_1, [v_1, v_2]) \ \wedge c = (a * v_1 + b * v_2, v_2) \end{aligned} \tag{25}$$

   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:

   $$\begin{aligned} &\underset{(v_1,v_2,v_3)}{argmax} && \mathcal{D}([v_1, v_2, v_3], oct_o) \\ &\text{subject to} && encode(oct_1, [v_1, v_2, v_3]) \ \vee encode(oct_2, [v_1, v_2, v_3]) \end{aligned} \tag{26}$$

   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. 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:

$$k_1 * BCELoss(e_i, 1 \mid e_i < 0.5) + k_2 * \sum_{i \mid e_i \geq 0.5} w_i \tag{27}$$

   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.