# ConstraintFlow: A Declarative DSL for Certified Artificial Intelligence

Anonymous Author(s)

## 1 PROBLEM AND MOTIVATION

Growing concerns about the trustworthiness of state-of-the-art Artificial Intelligence (AI) systems hinder their deployment in safety-critical domains such as autonomous driving [4] and healthcare [2]. To tackle this, the field of Certified AI aims to formally certify the trustworthiness of Deep Neural Networks (DNNs). Formal guarantees provide a more reliable metric for assessing the suitability of a DNN for real-world deployment than standard test-set accuracy. Unfortunately, existing certifiers are severely limited in the types of models, properties, and hardware that they can handle. Extending these works to new cases is often error-prone and requires substantial manual effort and expertise, limiting their accessibility to the general users of deep learning frameworks who may not have the necessary background to develop a DNN certifier.

To overcome these barriers, we envision a compiler framework that can automatically generate precise, scalable, fast, and memory-efficient code for certifying properties for arbitrary DNNs, optimized for different hardware. As the first step, we propose a DSL called *ConstraintFlow* to specify DNN certifiers with minimal high-level description. For example, Figure 1 shows that we can specify thousands of lines of intricate C-code of DeepPoly [1] in just 16 lines of ConstraintFlow. The certifier specifications in ConstraintFlow are functional, declarative, and pointer-free making it suitable for advanced compiler optimizations. Further, we also support automatic verification for the correctness of the certifier code to eliminate any algorithmic or implementation errors early in the development process.

## 2 BACKGROUND AND RELATED WORK

Most state-of-the-art DNN certifiers achieving the best tradeoff between precision and scalability, are based on Abstract Interpretation [5]. At a high level, these certifiers have 3 main steps: (i) designing an abstract domain whose elements should over-approximate the values of all the neurons, (ii) developing abstract transformers that should over-approximate the effect of each DNN operation like Affine, ReLU, etc, and (iii) proving that the abstract elements and transformers over-approximate the concrete neuron values.

The domains used in state-of-the-art certifiers associate relational (e.g., DeepPoly, DeepZ) or non-relational (e.g., Interval) constraints with each neuron. For example, for each neuron $n$, DeepPoly [9] shape associates four constraints - $\langle l, u, L, U\rangle$ where $L$ and $U$ are polyhedral expressions of the form $\sum_j a_j \cdot n_j$ s.t. $L \leq n \leq U$, and $l$ and $u$ are interval constraints, s.t., $l \leq n \leq u$. Similarly, Zonotope-based certifiers [3, 7, 8] associate with each neuron an affine form $Z_n = \sum_j a_j \cdot \epsilon_j$, where $\forall j,\ \exists\, \epsilon_j \in [-1, 1]$ s.t. $Z_N = n$. The abstract transformers compute the constraints for each neuron based on the constraints of previously visited neurons. Finally, one proves that the transformers are *sound*, i.e., they meet the over-approximation property.

## 3 APPROACH AND UNIQUENESS

We describe ConstraintFlow next.

**Abstract elements** Different expressions used by the popular domains like DeepPoly and DeepZ are provided as different types (`PolyExp` and `ZonoExp`) in ConstraintFlow. We define a `Shape` construct to specify a set of expressions that define the abstract element for each neuron. For instance, the DeepPoly shape can be defined in ConstraintFlow as in Line 1 Figure 1a.

**Abstract Transformers** To specify transformers in ConstraintFlow, we provide a `transformer` construct which encapsulates the computations for each expression of the abstract element in a functional and declarative style. Further, we provide a `flow` construct to specify the order of the flow of constraints in the DNN. This includes forward flow (input layer to output layer), backward flow or any sequence of these [10]. This construct instructs the compiler to apply the computation specified in the `transformer` to all the neurons in the DNN in the specified order. For instance, in line 16 of code 1a, the **`deeppoly`** transformer is applied to the neurons of the DNN in a layer-wise fashion traversing the DNN in the forward direction until the last layer.

Depending on the flow direction, the computations for a neuron depend on the neurons in previous layers (for forward flow) or next layers (backward flow). These neurons can be accessed using the **`prev`** construct. For example, line 9 of the code in Figure 1a shows the DeepPoly approximation of the ReLU operator in ConstraintFlow. In the case of ReLU, **`prev`** is a singleton list. The code computes the convex approximation in 3 cases depending on the bounds of the neurons in the `prev` list. The functional and declarative style of the specifications will lead to effective compiler optimizations and parallelizations. For instance, in the case of Figure 1a, since the neuron ordering within each layer is not specified, the compiler can take the opportunity to parallelize their computations.

The Affine abstract transformer for DeepPoly (line 13 In Figure 1a) involves a more complicated *backsubstitution* step for computing the $l$ and $u$ for the neurons. Given a polyhedral expression, this step iteratively visits its constituent neurons in a specific order (possibly different from the one specified in `flow`), and replaces them with their respective bounds. The order affects the precision and the speed of the backsubstitution step. In ConstraintFlow, we provide `traverse` construct for a directed traversal through the DNN where the order can be specified by the programmer. This provides a flexible way of specifying traversal without having to write any loops, which hinder compiler optimizations and automatic verification. The programmer can use the flexibility provided by `traverse` to adjust the precision/speed tradeoff. Lines 7 and 8 in Figure 1a shows how `traverse` can be used for the backsubstitution step in DeepPoly.

**Verification** To ensure soundness, the abstract shape is also associated with a soundness condition that the transformers need to satisfy. For example, for DeepPoly, each neuron's value should lie

```
1  def Shape as (Float l, Float u, PolyExp L, PolyExp U){(curr[l]<=curr) and (curr[L]<=curr) and (curr[u]>=curr) and (curr[U]>=curr)}
2  func simplify_lower(Neuron n, Float c) = (c>=0) ? (c*n[l]) : (c*n[u]);
3  func simplify_upper(Neuron n, Float c) = (c>=0) ? (c*n[u]) : (c*n[l]);
4  func replace_lower(Neuron n, Float c) = (c>=0) ? (c*n[L]) : (c*n[U]);
5  func replace_upper(Neuron n, Float c) = (c>=0) ? (c*n[U]) : (c*n[L]);
6  func backsubs_lower(PolyExp e, Neuron n) = (e.traverse(backward, -n[layer], true, replace_lower){e<=n}).map(simplify_lower);
7  func backsubs_upper(PolyExp e, Neuron n) = (e.traverse(backward, -n[layer], true, replace_upper){e>=n}).map(simplify_upper);
8  transformer deeppoly(curr, prev){
9      Relu -> sum(prev[l])>=0 ? (sum(prev[l]), sum(prev[u]), sum(prev[L]), sum(prev[U])) :
10     (sum(prev[u])<=0 ? (0,0,0,0) : (0, sum(prev[u]), 0,
11     (sum(prev[u])/(sum(prev[u])-sum(prev[l])))*sum(prev)-(sum(prev[u])*sum(prev[l])/(sum(prev[u])-sum(prev[l])))))
12
13     Affine -> (backsubs_lower(prev.dot(curr[weight]) + curr[bias], curr), backsubs_upper(prev.dot(curr[weight]) + curr[bias], curr),
14     prev.dot(curr[weight]) + curr[bias], prev.dot(curr[weight]) + curr[bias])
15 }
16 flow(forward, n[layer], true, deeppoly)
```

**(a) DeepPoly certifier in just 16 lines of ConstraintFlow**

```
161  elina_abstract0_t* fpp
162          size_t intdim
163          double * lexpr
164          double * uexpr
165          size_t expr_s
166          size_t spatial
167      fppoly_t * res =
168
169          fppoly_from_ne
170          size_t num_pix
171          res->input_lex
172          res->input_uex
```

```
628      assert(num_pred
629      fppoly_t *fp =
630      size_t numlayer
631      size_t pred = p
632      size_t num_in_n
633      size_t num_out_
634      fppoly_add_new_
635
636      size_t i, j, k;
637
638      neuron_t **out_
639      k = 0;
```

```
1208     fppoly_t *fp = fppoly_of_abstract0(abs);
1209     if(layerno >= fp->numlayers){
1210         fprintf(stdout,"the layer does not exist\n");
1211         return;
1212     }
1213     if(!fp->layers[layerno]->is_activation){
1214         fprintf(stdout, "the layer is not an activation layer\n");
1215         return;
1216     }
1217     layer_t * layer = fp->layers[layerno];
1218     neuron_t * neuron = layer->neurons[neuron_no];
1219     free_expr(neuron->lexpr);
1220     neuron->lexpr = NULL;
```

**(b) Thousands of lines of intricate C code taken from just one of the many (around 10) files of DeepPoly Specification in the ELINA framework**

**Figure 1: Over 1000x Lines of Code (LOC) improvement from C to ConstraintFlow**

between its bounds, i.e., $l \leq n \leq u, L \leq n \leq U$. In ConstraintFlow, we provide constructs to specify such soundness conditions (as in line 1 Figure 1a). We provide automatic verification of the soundness of abstract transformers w.r.t these conditions to reduce the burden of proving them manually. The minimal, declarative, and functional design of ConstraintFlow makes automatic verification efficient.

## 4 RESULTS AND CONTRIBUTION

We have implemented the lexer, parser, type-checking, and automated verification in Python. We can verify the correct implementations of Interval, DeepZ, and DeepPoly certifiers, written in $< 20$ LOC, by automatically generating Z3 [6] queries from their ConstraintFlow implementations. We can identify bugs in any incorrect implementations of these algorithms. All verification queries finish within $< 0.2s$. Our design makes writing new certifiers substantially simpler allowing developers to focus on designing better abstract domains and abstract transformers, without manually checking their soundness.

# REFERENCES


[1] [n.d.]. ELINA: ETH LIbrary for Numerical Analysis. http://elina.ethz.ch/. Accessed: 2023-22-03.

[2] Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. 2013. Artificial neural networks in medical diagnosis. *Journal of Applied Biomedicine* 11, 2 (2013).

[3] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and Abstraction: A Synergistic Approach for Analyzing Neural Network Robustness. In *Proc. Programming Language Design and Implementation (PLDI)*. 731–744.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

[5] PATRICK COUSOT and RADHIA COUSOT. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation* 2, 4 (08 1992), 511–547. https://doi.org/10.1093/logcom/2.4.511 arXiv:https://academic.oup.com/logcom/article-pdf/2/4/511/2740133/2-4-511.pdf

[6] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340.

[7] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 3–18. https://doi.org/10.1109/SP.2018.00058

[8] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and effective robustness certification. *Advances in Neural Information Processing Systems* 31 (2018), 10802–10813.

[9] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[10] Haoze Wu, Clark Barrett, Mahmood Sharif, Nina Narodytska, and Gagandeep Singh. 2022. Scalable Verification of GNN-Based Job Schedulers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 162 (oct 2022), 30 pages. https://doi.org/10.1145/3563325