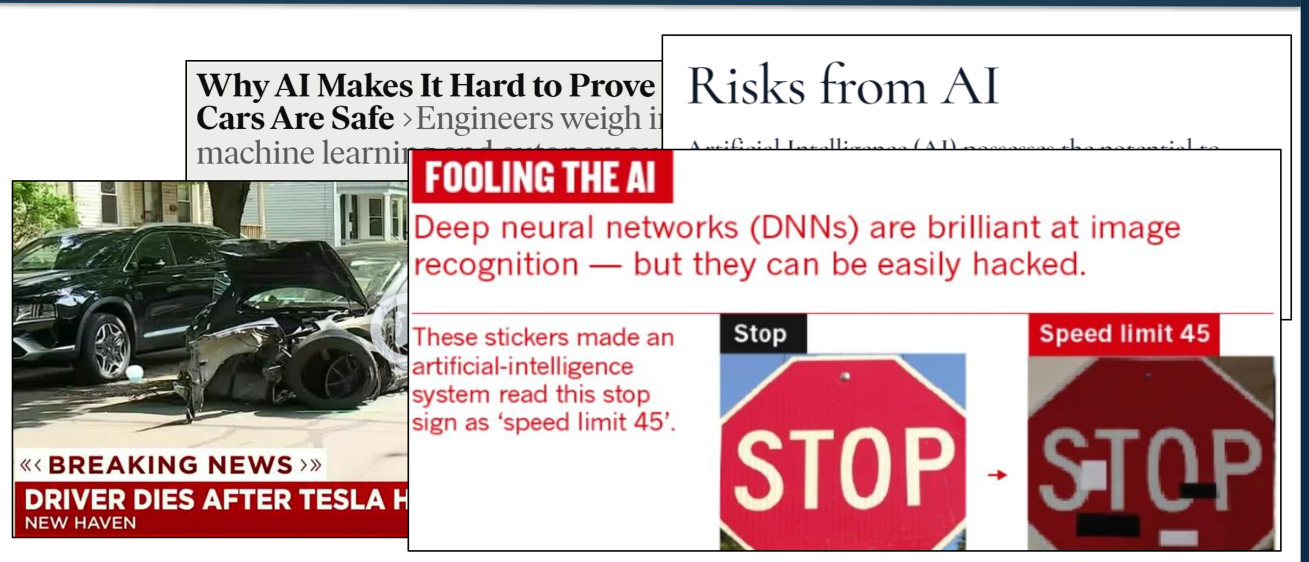
ConstraintFlow: A Declarative DSL for Certified Artificial Intelligence

Avaljot Singh

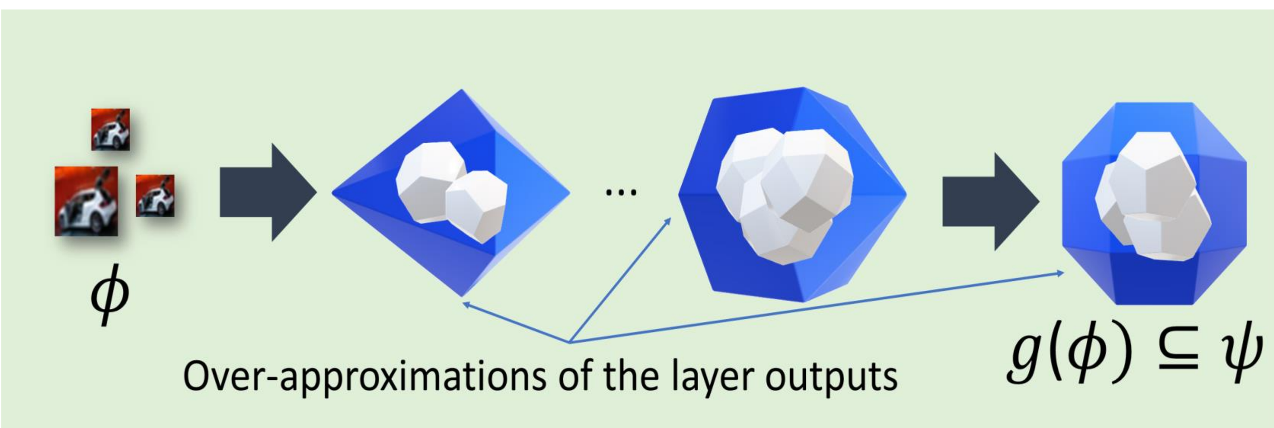
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Unreliable AI



Standard test set accuracy is not enough.
Formal guarantees provide a more reliable metric

Formal Certification

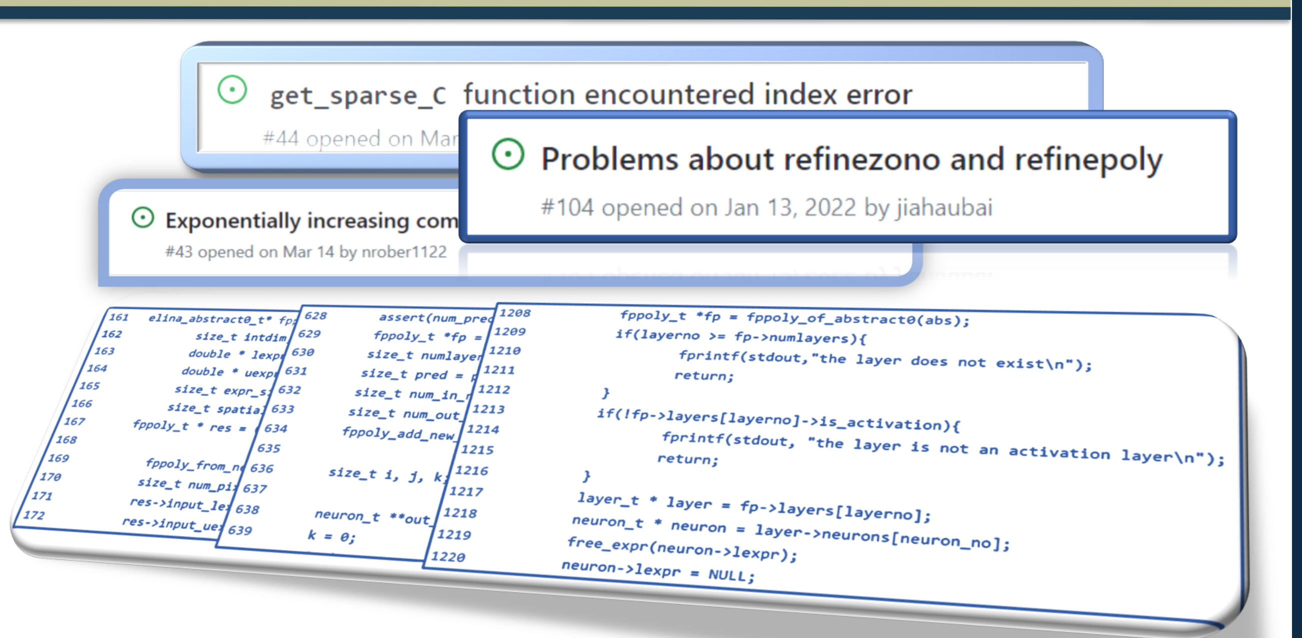


Certification using Abstract Interpretation

Abstract Domains

1. Polyhedral – DeepPoly, CROWN, Fast-Lin, Neurify
2. Zonotopes – DeepZ, RefineZono, AI2
3. Symbolic Intervals – NeuroDiff, ReluDiff, ReluVal

Problems with Existing Libraries



Enormous unverified codebases, error-prone, non-scalable, limited DNN architecture

ConstraintFlow Design

Over-approximation-based Soundness	• Abstract Interpretation based DNN certifiers need to maintain over-approximation. Existing libraries can be buggy. We provide bounded automatic verification to ensure correctness of **soundness property**.
Polyhedral Expressions	• DNN certifiers have large polyhedral expressions which are implemented using large arrays involving intricate pointer arithmetic. We provide these as separate types - **PolyExp**
Complex Imperative Code	• Existing libraries are implemented using complex imperative constructs like nested loops, making verification hard. We provide **declarative, loop-free, pointer-free** DSL design.
Arbitrary Graph Traversal	• Transformers need arbitrary graph traversal through the DNN, which makes verification hard. We provide constructs for **user defined Invariants** to support lightweight automated verification.
Correctness of Semantics	• Both operational and verification semantics have symbolic variables which is non-trivial. We provide **type-checking rules** and prove **theorems** to support the correctness of **verification procedure** w.r.t **operational semantics**.

```
def Shape as (Float l, Float u, PolyExp L, PolyExp U){(curr[l]<=curr) and (curr[u]>=curr) and (curr[L]<=curr) and (curr[U]>=curr)};

func simplify_lower(Neuron n, Float c) = (c >= 0) ? (c * n[l]) : (c * n[u]);

func simplify_upper(Neuron n, Float c) = (c >= 0) ? (c * n[u]) : (c * n[l]);

func replace_lower(Neuron n, Float c) = (c >= 0) ? (c * n[L]) : (c * n[U]);

func replace_upper(Neuron n, Float c) = (c >= 0) ? (c * n[U]) : (c * n[L]);

func priority(Neuron n) = n[layer];

func backsubs_l (PolyExp e, Neuron n) = (e.traverse(backward, priority, true, replace_lower) {e <= n}).map(simplify_lower);

func backsubs_u (PolyExp e, Neuron n) = (e.traverse(backward, priority, true, replace_upper) {e >= n}).map(simplify_upper);

transformer deeppoly(curr, prev){

  Affine -> (backsubs_lower(prev.dot(curr[weight]) + curr[bias], curr),
             backsubs_upper(prev.dot(curr[weight]) + curr[bias], curr),
             prev.dot(curr[weight]) + curr[bias], prev.dot(curr[weight]) + curr[bias]);

}

flow(forward, -priority, true, deeppoly);
```

TYPE-CHECKING TRAVERSE

$$\frac{\begin{array}{c}\Gamma, \tau_s \vdash e_1 : PolyExp \quad \Gamma, \tau_s \vdash e_2 : Ct \\ \Gamma, \tau_s \vdash f_{c_1} : Neuron \rightarrow \gamma' \quad \Gamma, \tau_s \vdash f_{c_2} : Neuron \rightarrow Bool \\ \Gamma, \tau_s \vdash f_{c_3} : Neuron \times Float \rightarrow \gamma \\ \gamma' \sqsubseteq Float \quad \gamma \sqsubseteq PolyExp\end{array}}{\Gamma, \tau_s \vdash e_1 \cdot traverse(d, f_{c_1}, f_{c_2}, f_{c_3})\{e_2\} : \gamma'}$$

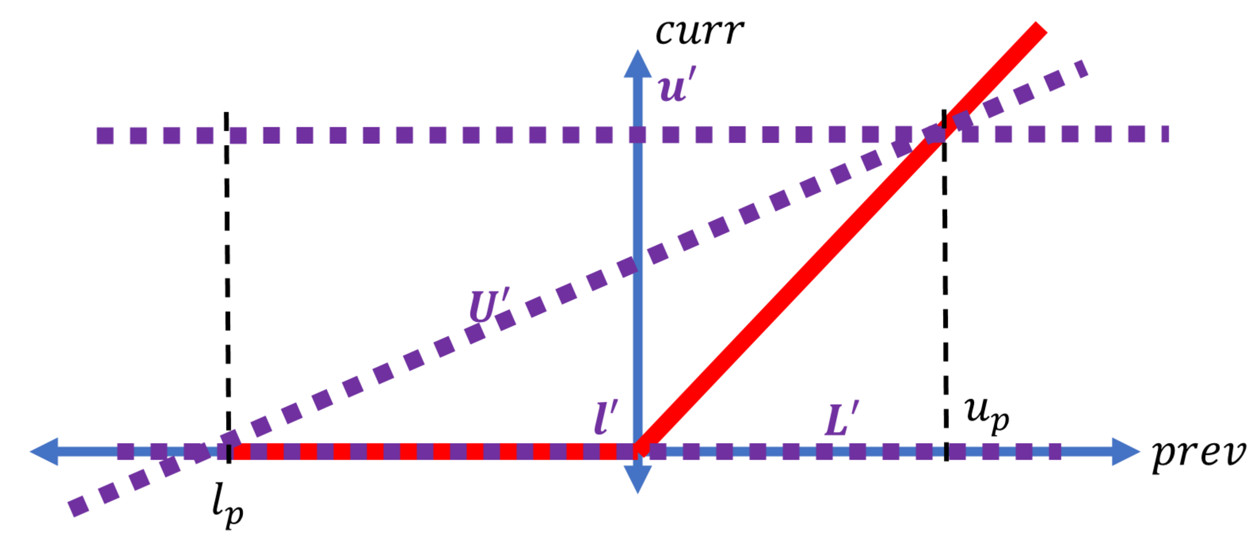
INVARIANT CHECK

$$\frac{\begin{array}{c}\langle e, F, \sigma, \mathcal{H}_S, C \rangle \downarrow \mu, C' \\ \mu_b = unsat(\neg(C' \Rightarrow \mu)) \\ \mu_b' = checkInduction(x \cdot traverse(d, f_{c_1}, f_{c_2}, f_{c_3})\{e\}), F, \sigma, \mathcal{H}_S, C)\end{array}}{checkInvariant(x \cdot traverse(d, f_{c_1}, f_{c_2}, f_{c_3})\{e\}), F, \sigma, \mathcal{H}_S, C) = \mu_b \wedge \mu_b', C'}$$

OPERATIONAL SEMANTICS TRAVERSE

$$\frac{\begin{array}{c}V' = P(V, f_{c_1}, F, \rho, \mathcal{H}_C) \qquad \nu = c + \nu_{V'} + \nu_{\overline{V}}, b \\ \langle \nu_{V'}.map(f_{c_3}), F, \rho, \mathcal{H}_C \rangle \Downarrow \nu' \\ V'' = \text{Filter}((V - V') \cup N(V', d), f_{c_2}, F, \rho, \mathcal{H}_C) \\ \langle \nu' \cdot traverse(d, f_{c_1}, f_{c_2}, f_{c_3})\{e\}\rangle, F, \rho, \mathcal{H}_C, V'') \Downarrow \nu'\end{array}}{\langle \nu \cdot traverse(d, f_{c_1}, f_{c_2}, f_{c_3})\{e\}\rangle, F, \rho, \mathcal{H}_C, V) \Downarrow \nu''}$$

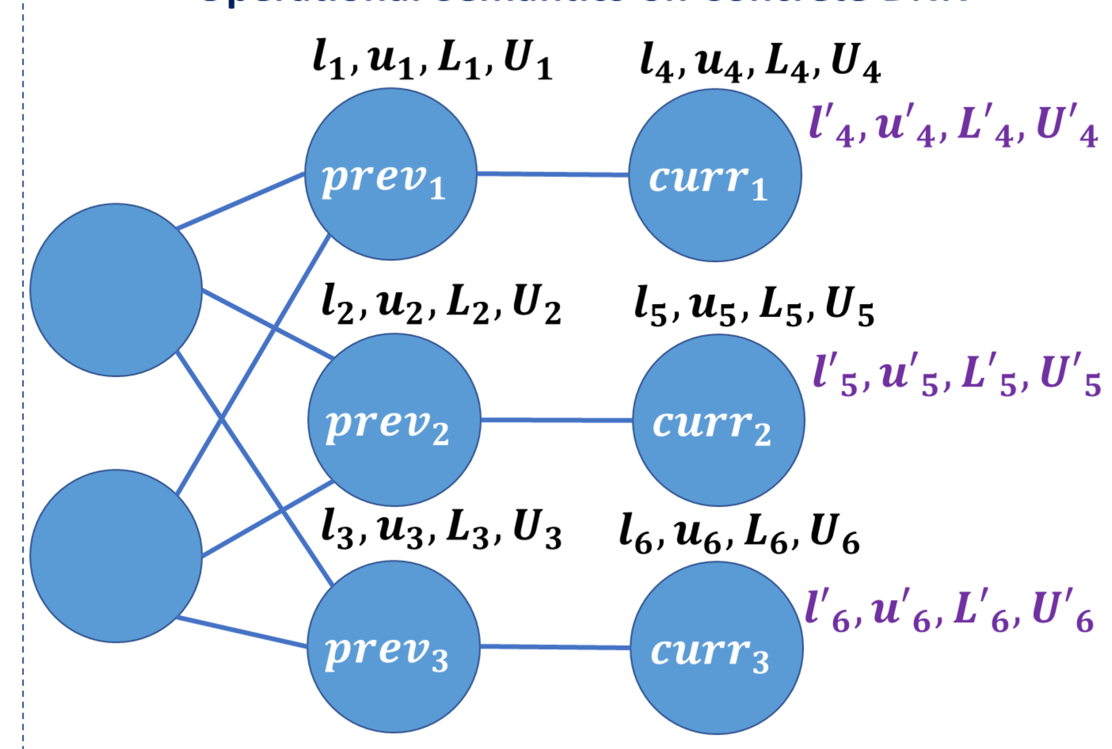
Bounded Automated Verification



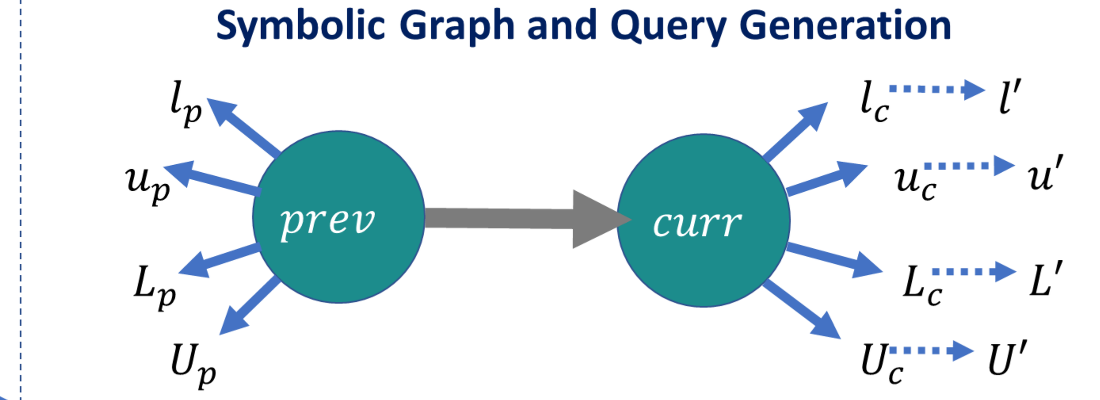
$$(l' \leq curr) \wedge (u' \geq curr) \wedge (L' \leq curr) \wedge (U' \geq curr)$$

```
transformer deeppoly(curr, prev){
    Relu -> prev[l] >= 0 ? (prev[l], prev[u], prev, prev) :
    (prev[u] <= 0 ? (0, 0, 0, 0) : (0, prev[u], 0,
    ((prev[u]/(prev[u]-prev[l]))*prev)-
    ((prev[u]*prev[l])/(prev[u]-prev[l]))));
}
```

Operational Semantics on Concrete DNN



Symbolic Graph and Query Generation



$$\phi = \wp(prev) \wedge \wp(curr) \wedge curr = ReLU(prev)$$
$$\psi = \wp(curr')$$
$$\phi \wedge (curr = curr') \Rightarrow \psi \text{ ? (SMT query)}$$

- **In ConstraintFlow, we can code the standard DNN certifiers in less than 20 LOCs.**
- **For the first time, we can verify the soundness of the existing certifiers on arbitrary DNNs that are bounded DAGs**
- **The certifier code in ConstraintFlow can be written for any hardware and DNN architecture and is decoupled from domain-specific optimizations.**

Main Theorems

Type-checking

- If an expression (e) type-checks to γ, under a context (Γ, τ_s) s.t. $(\bot \sqsubset \gamma \sqsubset \top)$ then under an environment $(F, \rho, \mathcal{H}_c)$ that is consistent with the context, it evaluates to a value (v) which is of the type γ' s.t. $\gamma' \sqsubseteq \gamma$.

$$(\Gamma, \tau_s \vdash e : \gamma) \wedge (\bot \sqsubset \gamma \sqsubset \top) \wedge (F, \rho, \mathcal{H}_c \sim \Gamma, \tau_s) \Rightarrow (\langle e, F, \rho, \mathcal{H}_c \rangle \Downarrow \nu) \wedge (\vdash \nu : \gamma') \wedge (\gamma' \sqsubseteq \gamma)$$

Over-approximation

- If the program (σ) type-checks, then the symbolic evaluation rules (verification procedure) correctly over-approximate the concrete operational semantics.

$$(\tau_s \vdash s : \Gamma) \wedge (\mathcal{H}_c \sim \tau_s) \Rightarrow \{s, \tau_s\} \rightsquigarrow (\mathcal{H}_s, C) \wedge \{| s, \tau_s, \mathcal{H}_c, \mathcal{H}_s, C |\} \updownarrow [v], [u] \wedge ([v], \mathcal{H}_C \preccurlyeq_C [u], \mathcal{H}_s)$$

Correctness

- If the property is proved on the transformer using bounded verification, then the DNN certifier is sound for any DNN that is a DAG within the bounds.

Evaluation

- We present the time in seconds (y-axis) to verify (timeout-300s) the standard correct abstract transformers and disprove incorrect ones averaged over a fixed set (size-12) of parameter values (max no. of parents and neurons in PolyExp). The percentage of the proved correct transformers or disproved incorrect transformers (within timeout) are shown.
- We can design and verify new transformers (* marked), like the reduced product of polyhedral and zonotope constraints.