Code2Inv: A Deep Learning Framework for Program Verification

Xujie Si et al.

August 19, 2020

Reference

[1] Learning Loop Invariants for Program Verification.
[2] Code2Inv: Learning Loop Invariants for Program Verification.

Background

Counterexample-guided Inductive Synthesis(CEGIS):
CEGIS is a classical paradigm for program verification where

- *Generator* proposs a candidate solution.
- *Checker* determines whether the solution is correct or not. If not, provide counterexample to refine later.

Shortcomes?

- The framework does not utilize the info that can be directly taken from program source.
- Candidate cannot be adaptive for the program.

Overview of CODE2INV

CODE2INV is an extensible end-to-end deep learing framework aims to compensate above shortcomes. The contribution of the tool is

- A framework for program verification which leverage deep learning, reinforcement learning and attention mechanism..
- Two small-scale instance of CODE2INV: loop invariant synthesizer and Constrained Horn Clause(CHC) Solver.

Idea of Invariant Synthesis

The basic idea of this paper is to mimic how human experts reason the loop invariant.

Example

```
1   int main() {
2    int x = 0, y = 0;
3    while (*) {
4      if (*) {
5        x++;
6        y = 100;
7      } else if (*) {
8        if (x >= 4) {
9          x++;
10         y++;
11       }
12       if (x < 0) y--;
13     }
14   }
15   assert( x < 4 || y > 2);
16 }
```

- Start by reading the assertion. x, y.
- x may increase. $x < 4$ may not always hold.
- How can $x \geq 4$: by executing the first branch 4 times. Then we guess $y \geq 100$ because...
- $x < 4 \vee y \geq 100$ is not enough, then add $x \geq 0$
- $x \geq 0 \wedge (x < 4 \vee y \geq 100)$

Idea of Invariant Synthesis

After going through the reasoning process three key components are concluded:

- Organize the program in a hierarchical structured way.
- Compose the loop invariant step by step.
- Focus on different part of the program at each step.

Preliminaries

Definition (Multilayer Perceptron(MLP))

A multilayer perceptron(MLP) is a basic neural network model approximating an arbitrary continuous function $\vec{y} = f^*(\vec{x})$, where $\vec{x}, \vec{y}$ are numeric vectors. An MLP defines a mapping $\vec{y} = f(\vec{x}; \theta)$.

Definition (RNN)

An RNN approximate the mapping from a sequence of inputs $\vec{x}^1, \ldots, \vec{x}^t$ to either a single output $\vec{y}$ or a sequence of outputs $\vec{y}^1, \ldots, \vec{y}^t$. It defines a mapping $\vec{h}^t = f(\vec{h}^{t-1}, \vec{x}^t; \theta)$.
A common RNN model are the long short-term memory network(LSTM) and gated recurrent units(GRUs).

Remark. GNN: commonly used to learn over graph structure data. It learns an embedding for each node of the given graph.

General Framework of CODE2INV

Domains of Program Structures:

$$
\begin{aligned}
\boldsymbol{\mathcal{G}}(T) &= G_{\text{inst}} \\
\boldsymbol{\mathcal{G}}(A) &= G_{\text{inv}} \\
A &= \langle \Sigma \uplus H, N, P, S \rangle \\
x &\in H \uplus N \\
v &\in \Sigma \\
n &\in N \\
p &\in P \\
& \quad S \\
inv &\in \boldsymbol{\mathcal{L}}(A) \\
cex &\in \mathbb{C} \\
C &\in \boldsymbol{\mathcal{P}}(\mathbb{C}) \\
check(T, inv) &\in \{\bot\} \uplus \mathbb{C}
\end{aligned}
$$

Domains of Neural Structures

$$
\begin{aligned}
\pi &= \langle \nu_{\text{T}}, \nu_{\text{A}}, \eta_{\text{T}}, \eta_{\text{A}}, \alpha_{\text{ctx}}, \epsilon_{\text{inv}} \rangle \\
& \quad d \\
\nu_{\text{T}},\ \eta_{\text{T}}(G_{\text{inst}}) &\in \mathbb{R}^{|G_{\text{inst}}| \times d} \\
\nu_{\text{A}},\ \eta_{\text{A}}(G_{\text{inv}}) &\in \mathbb{R}^{|G_{\text{inv}}| \times d} \\
ctx &\in \mathbb{R}^d \\
state &\in \mathbb{R}^d \\
\alpha_{\text{ctx}} &\in \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d \\
\epsilon_{\text{inv}} &\in \boldsymbol{\mathcal{L}}(A) \to \mathbb{R}^d \\
aggregate &\in \mathbb{R}^{\text{k} \times d} \to \mathbb{R}^d \\
\nu_{\text{A}}[n] &\in \mathbb{R}^{\text{k} \times d} \\
\nu_{\text{T}}[h] &\in \mathbb{R}^{\text{k} \times d}
\end{aligned}
$$

General Framework of Code2Inv

Definition (Neural Policy)

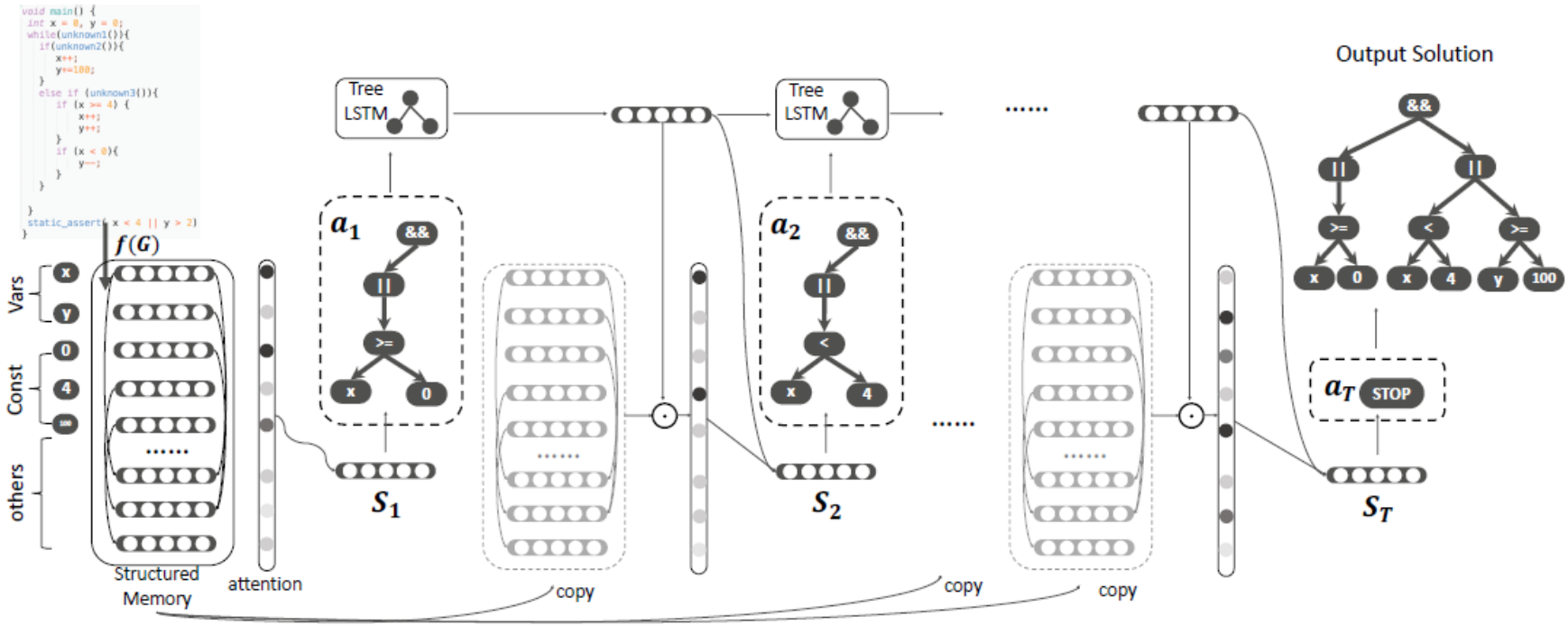
The neural policy is the key component of the framework and is define as

$$\pi = \langle \nu_T, \nu_A, \eta_T, \eta_A, \alpha_{ctx}, \epsilon_{inv} \rangle$$

Neural Policy is the most important component of the framework where

- η_T, η_A are two graph neural networks used for computing the neural embeddings ν_T, ν_A for graph representations G_{inst}, G_{inv} respectively.
- α_{ctx} is a neural network implemented as GRU maintains the attention context *ctx* when modifying invariant.
- ϵ_{inv} implemented as a Tree-LSTM is an invariant encoder that encodes partial generated invariant into *state*, which is used to update the attention.

Overall Framework of Loop Invariant Synthesis



General Framework of CODE2INV

Algorithm 1. Code2Inv Framework

Input: a verification instance T and a proof checker *check*
Output: a invariant *inv* satisfying $check(T, inv) = \bot$
Parameter: graph constructor $\mathcal{G}$ and invariant grammar A

```
1  π ← initPolicy(T, A)
2  C ← ∅
3  while true do
4      inv ← sample(π, T, A)
5      ⟨π, C⟩ ← improve(π, inv, C)
6  Function initPolicy(T, A)
7      Initialize weights of η_T, η_A, α_ctx, ε_inv with random values
8      ν_T ← η_T(G(T))
9      ν_A ← η_A(G(A))
10     return ⟨ν_T, ν_A, η_T, η_A, α_ctx, ε_inv⟩
```

Structured External Memory

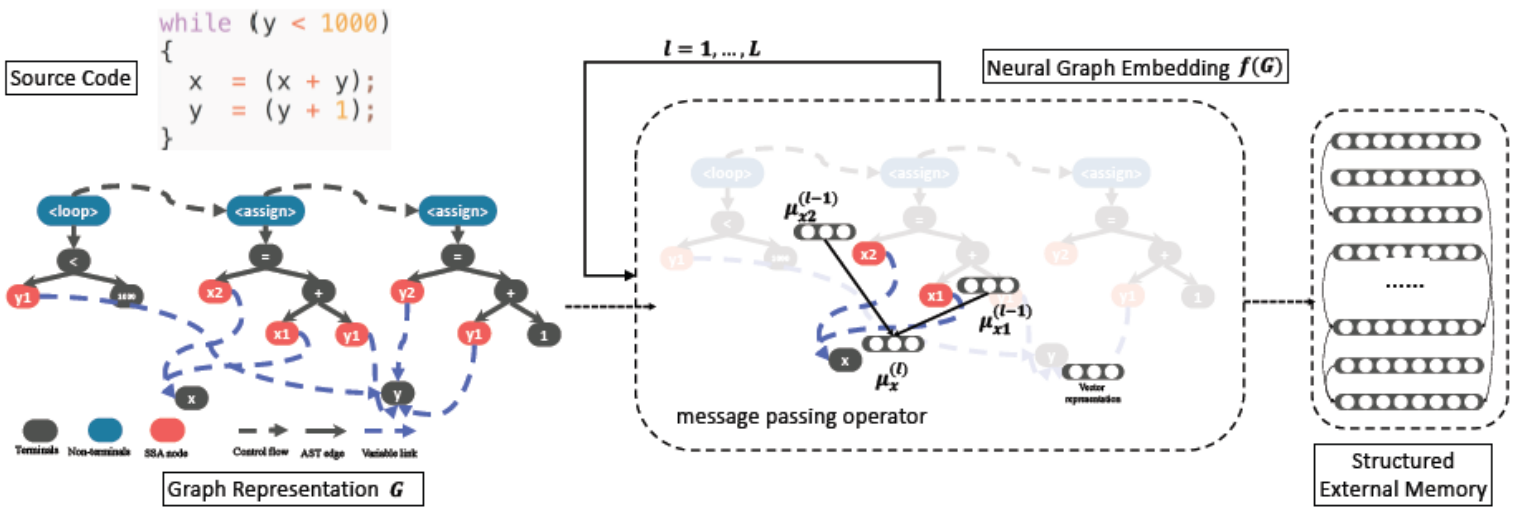
- First, convert a given program into a static single assignment form, then a flow graph where a vertex is an AST.
- Then, convert the graph into vector representation.
  $E = \{(e_x^{(i)}, e_y^{(i)}, e_t^{(i)})\}$.

$$\mu_v^{(l+1)} = h(\{\mu_u^{(l)}\}_{u \in \mathcal{N}^k(v), k \in \{1,2,\ldots,K\}})$$

$$\mu_v^{(0)} = \mathbf{W}_1 \mathbf{x}_v$$

$$\mu_v^{(l+1),k} = \sigma(\textstyle\sum_{u \in \mathcal{N}^k(v)} \mathbf{W}_2 \mu_u^{(l)}), \forall k \in \{1,2,\ldots,K\}$$

$$\mu_v^{(l+1)} = \sigma(\mathbf{W}_3 [\mu_v^{(l+1),1}, \mu_v^{(l+1),2}, \ldots, \mu_v^{(l+1),K}])$$



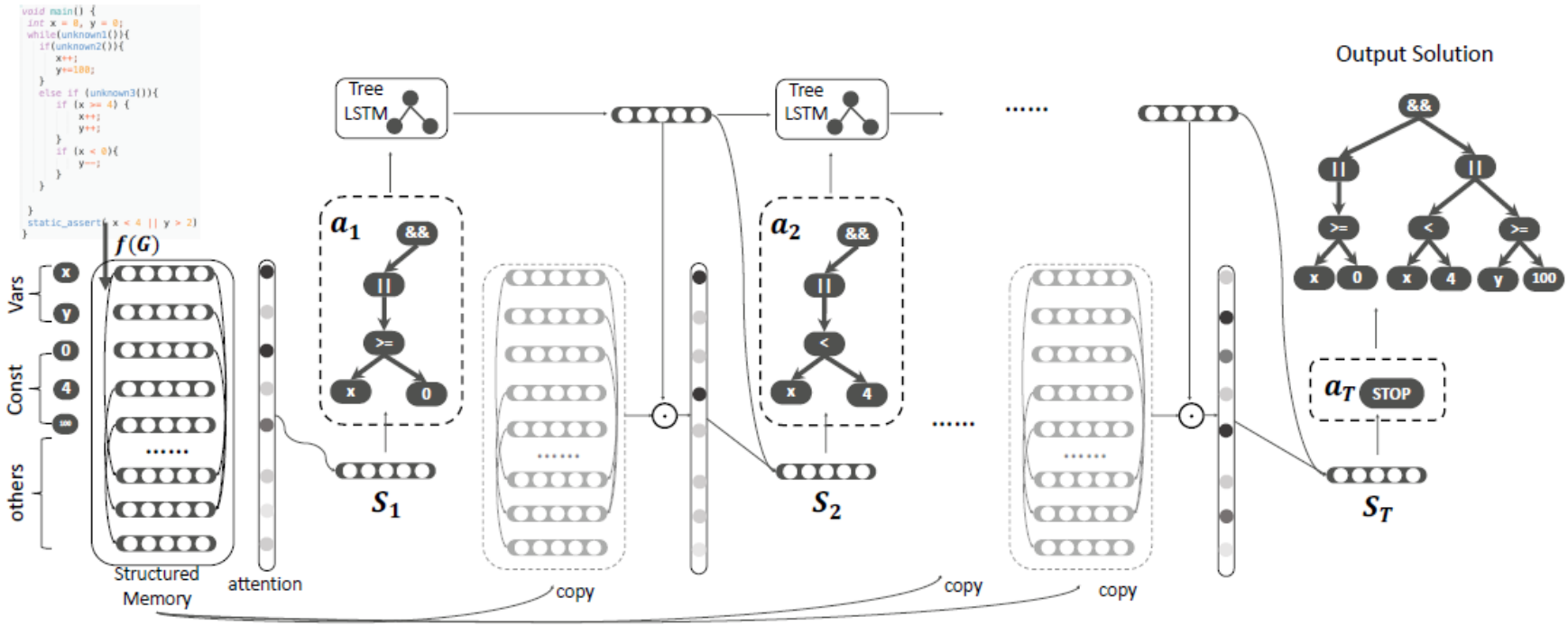
General Framework of CODE2INV

```
11 Function sample(π, T, A)
12     inv ← A.S
13     ctx ← aggregate(π.ν_T)
14     while inv is partially derived do
15         x ← leftmost non-terminal or placeholder symbol in inv
16         state ← π.ε_inv(inv)
17         ctx ← π.α_ctx(ctx, state)
18         if x is non-terminal then
19             p ← attention(ctx, π.ν_A[x], G(A))
20             expand inv according to p
21         else
22             v ← attention(ctx, π.ν_T[x], G(T))
23             replace x in inv with v
24     return inv

40 Function attention(ctx, ν, G)
41     Return node t in G such that dot product of ctx and ν[t]
42     is maximum over all nodes of G
```

General Framework of CODE2INV

```
25 Function improve(π, inv, C)
26     n ← number of counter-examples C that inv can satisfy
27     if n = |C| then
28         cex ← check(T, inv)
29         if cex = ⊥ then
30             save inv and weights of π
31             exit                                   // a sufficient inva
32         else
33             C ← C ∪ {cex}
34     r ← n/|C|
35     π ← updatePolicy(π, r)
36     return ⟨π, C⟩

37 Function updatePolicy(π, r)
38     Update weights of π.η_T, π.η_A, π.α_ctx, π.ε_inv, π.ν_T, π.ν_A by
39     standard policy gradient [34] using reward r
```

Overall Framework of Loop Invariant Synthesis



Multi-Step Decision Making Process

Definition (Loop Invariant)

We define the loop invariant to be a tree $\mathcal{T}$

$$\mathcal{T} = (\mathcal{T}_1 \vee \mathcal{T}_2 \ldots) \wedge (\mathcal{T}_{t+1} \vee \mathcal{T}_{t+2} \ldots) \wedge \ldots \wedge (\ldots \mathcal{T}_{T-1} \vee \mathcal{T}_T)$$

T_t is $X < 2 \times y + 10 - z$.
MDP: use MDP to model the problem of constructing the invariant incrementally.
$\mathcal{M}^G = (s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$

- $a_t = (op_t, \mathcal{T}_t)$. op_t can be $\wedge$ or $\vee$.
- $s_t = (G, \mathcal{T}^{<t})$.

Reward Design r_t

Early Reward: the goal is to quickly remove the trivial and meaningless predicates. e.g. $e == e, e < e$. Examine partially generated $\mathcal{T}^t$.

$$\frac{P \Rightarrow I \;\; (pre) \qquad \{I \wedge B\}\, S\, \{I\} \;\; (inv) \qquad (I \wedge \neg B) \Rightarrow Q \;\; (post)}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{Q\}}$$

Continuous Reward: $ce_{pre}, ce_{inv}, ce_{post}$ and $pass_{pre}, pass_{inv}, pass_{post}$. No new counterexample is introduced:

$$\frac{|pass_{pre}|}{|ce_{pre}|} + \frac{|pass_{inv}|}{|ce_{inv}|} + \frac{|pass_{post}|}{|ce_{post}|}$$

New counterexample introduced:

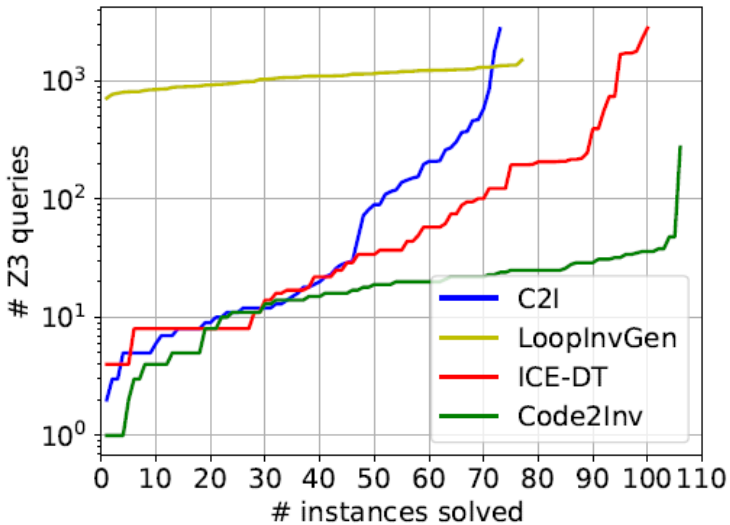
$$\frac{|pass_{pre}|}{|ce_{pre}|} + [pass_{pre} = ce_{pre}]\frac{|pass_{inv}|}{|ce_{inv}|} + [pass_{pre} = ce_{pre}][pass_{inv} = ce_{inv}]\frac{|pass_{post}|}{|ce_{post}|}$$

Termination

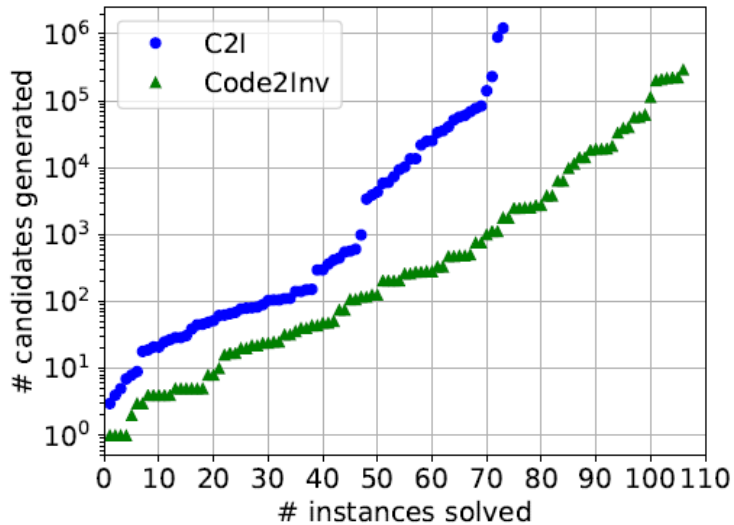
Situations for termination:

- Successfully generated.
- The tree of the generated invariant reach a ceiling number of branches.
- The agent generate invalid action.

Experimental Result



(a) verification cost by each solver



(b) sample complexity of C2I and CODE2INV

Ablation Study

Table 1: Ablation study for different configurations of CODE2INV.

configuration	#solved instances	max #Z3 queries	max #parameter updates
without CE, without attention	91	415K	441K
without CE, with attention	94	147K	162K
with CE, without attention	95	392	337K
with CE, with attention	106	276	290K
LSTM embedding + CE + attention	93	32	661K