Code2Inv: Learning Loop Invariants for Program Verification

Xujie Si*, Hanjun Dai*, Mukund Raghothaman, Mayur Naik, Le Song (*equal contribution)

Penn — Georgia Institute of Technology

Background

"Program testing can be used to show the presence of bugs, but never to show their ***absence***!"
— Edsger W. Dijkstra

Program Verification

"If P holds before executing S, then Q holds afterwards."

```
assume(y > 0)        | assume(x <= n)       | assume(x < 0)        {P}
if (x < 0){          | while (x < n){       | while (x < 0){
    x = y            |     x = x + 1        |     x = x + y        S
}                    | }                    |     y = y + 1
                     |                      | }
assert(x >= 0)       | assert(x == n)       | assert(y > 0)        {Q}
```

How to prove $\{P\}\ S\ \{Q\}$?

"Loop Invariant"

E.g. for the last program, $\boldsymbol{I} = x < 0 \vee y > 0$

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

[Hoare 1969]

Why is this interesting?

- Proving $\{P\}\ S\ \{Q\}$ requires deep logical reasoning
- Finding $\boldsymbol{I}$ is a fundamental problem in program verification
- Traditionally, ad hoc features are used to find $\boldsymbol{I}$
- Challenge problem in Artificial Intelligence

Problem Statement

1. How to find $\boldsymbol{I}$ in order to prove $\{P\}\ \textbf{\textit{while}}\ B\ \textbf{\textit{do}}\ S\ \{Q\}$?

2. Given a set of programs $\{S_i\} \sim \mathcal{P}$ that are sampled from an unknown distribution $\mathcal{P}$, can we ***learn*** from them and ***generalize*** the learned strategy to unseen programs?

Challenges

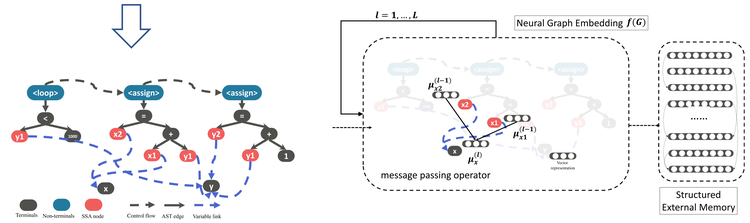
How to ***represent programs*** so that a neural policy can be learned?

How to ***predict loop invariant*** based on the neural representation?

How to ***learn a neural policy*** from an automated theorem prover?

Representing programs using ASTs and Dataflow

```
x1 = φ(x0, x2)                          while (y < 1000) {
y1 = φ(y0, y2)          SSA                 x = x + y
while (y1 < 1000) {                         y = y + 1
    x2 = x1 + y1                        }
    y2 = y1 + 1
}
```

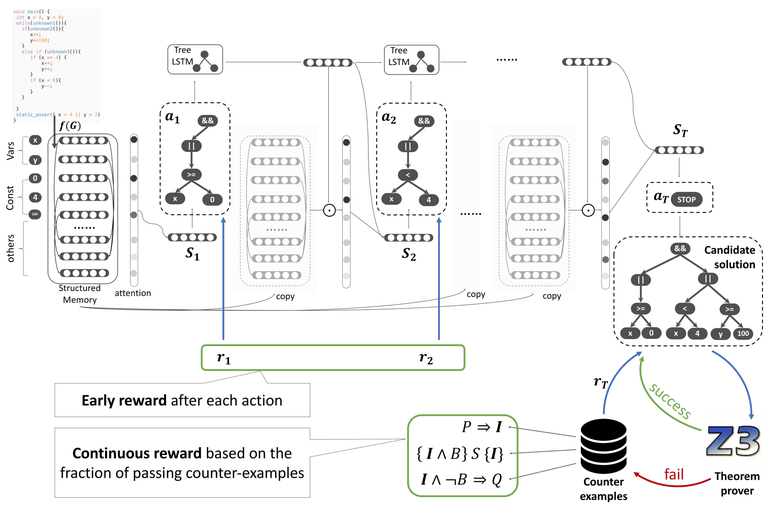


Loop Invariant Prediction Framework

Multi-step decision making process

Loop invariant | Program embedding | Partial loop invariant | Logical operator | Predicate AST

$$\pi(\mathcal{T}|G) = \prod_{t=1}^{T} \pi(a_t|\mathcal{T}^{(<t)}, G) = \prod_{t=1}^{T} \pi(op_t, \mathcal{T}_t|\mathcal{T}^{(<t)}, G)$$



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

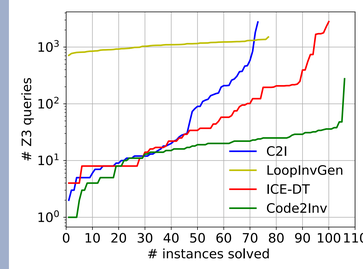
Experiments

Collected **133** benchmark programs (SyGuS)

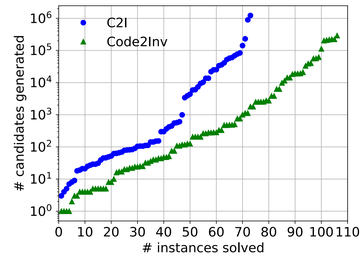
Compared our framework Code2Inv with 3 SOTA approaches

Code and data: https://github.com/PL-ML/code2inv

Higher performance even without training



Verification cost

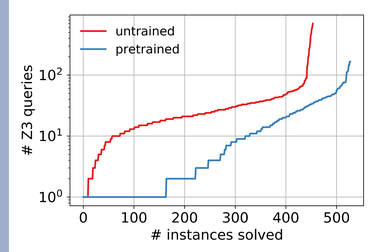


Sample complexity

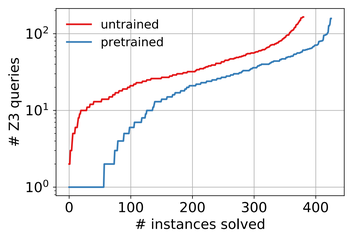
Attention and CE improve performance

Counter-example	Attention	#Solved Instances	Max #Z3 Queries	Max #Parameter Updates
✗	✗	91	415K	441K
✗	✓	94	147K	162K
✓	✗	95	392	337K
✓	✓	106	276	290K
LSTM + CE + Attention		93	32	661K

Successfully generalizes to new programs



With 1 confounding variable



With 5 confounding variables

Visualizing attention over code

```
int main()
{
  int n, k, c = 0;
  assume (n > 0);

  while(*) {
    if(c < n) {
      c = c + 1;
      k = 2;
    }
    if(c == n) {
      c = 1;
      k = 5;
    }
  }

  if(c == n) {
    assert (n > -1);
  }
}
```

attention for a == b

```
int main()
{
  int a = 0, b = 0, c = 0;

  while(*) {
      a += 1;
      b += 1;
      c -= 1;
  }

  if(a != b) {
    assert (b == -1);
  }
  else{
  }
}
```

attention for c >= -1 && n >= 1

University of Pennsylvania & Georgia Institute of Technology

Contact: {xsi,rmukund,mhnaik}@cis.upenn.edu; {hanjun.dai, lsong}@cc.gatech.edu