Learning Loop Invariants for Program Verification

Xujie Si et al.

May 26, 2020

Loop Invariant

Hoare Triple: $\{P\}S\{Q\}$
Rule about loops in Hoare's logic:

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

Loop Invariant Inference Problem: given a pre-condition P and a post-condition Q and a program S containing a single loop, can we find a predicate I such that $\{P\}S\{Q\}$ is valid?
Loop Invariant Inference Problem is undecidable.

Loop Invariant: Example

Example

$$x := -50; \texttt{while}\ (x < 0)\ \texttt{do}$$

$$\{x := x + y; y := y + 1\}\ \texttt{assert}\ (y > 0)$$

For the example loop above, it is easy to check $x < 0 \vee y > 0$ is a valid invariant.

$$x = -50 \rightarrow x < 0 \vee y > 0$$

$$x < 0 \vee y > 0 \wedge x \geq 0 \rightarrow y > 0$$

Idea

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

Example

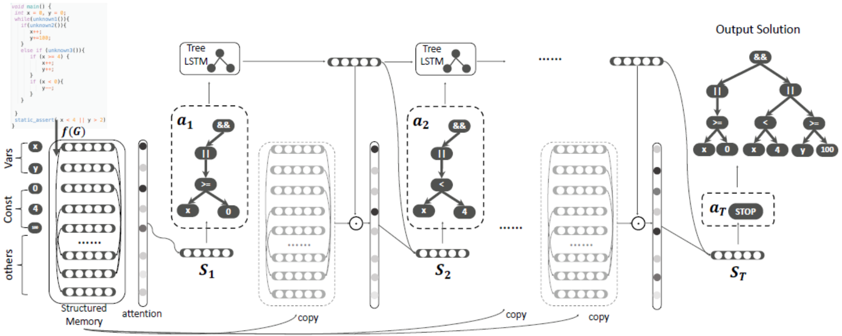
```
int main() {
  int x = 0, y = 0;
  while (*) {
    if (*) {
      x++;
      y = 100;
    } else if (*) {
      if (x >= 4) {
        x++;
        y++;
      }
      if (x < 0) y--;
    }
  }
  assert( x < 4 || y > 2);
}
```

- 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 $x \geq 0$
- $x \geq 0 \wedge (x < 4 \vee y \geq 100)$

Programming the Reasoning Procedure with NN

- A structured external memory representation of the program.
- Multi-step autoregressive model for incremental loop incariant construction.
- An attention component that mimics the varying focus in each step.

Overall Framework



Structured External Memory

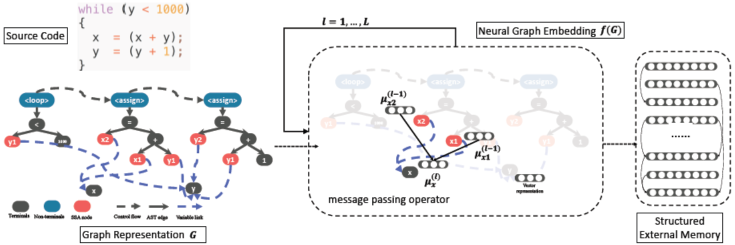
- First, convert a given program into a static single assignment form, then a flow graph where a vertex is anAST.
- Then, convert the graph into vector representation. $E = \{(e_x^{(i)}, e_y^{(i)}, e_t^{(i)})\}$.

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

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

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

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



Multi-Step Decision Making Process

Definition (Loop Invariant)

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

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

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

- $a_t = (op_t, \mathcal{T}_t)$. 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 $\mathcal{T}^t$.

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:

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

Training Objective and Expected Policy Reward

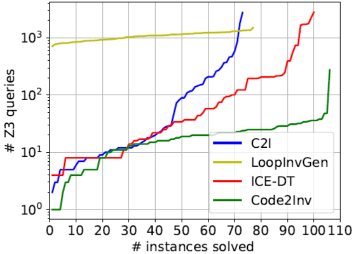
$$\max_{\theta,\phi} \mathbb{E}_{\pi(op_t,\mathcal{T}_t|\mathcal{T}^{(<t)},G;\phi)}\Big(\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'} - b(\mathcal{T}^{(<t)},G;\psi)\Big)$$

Termination

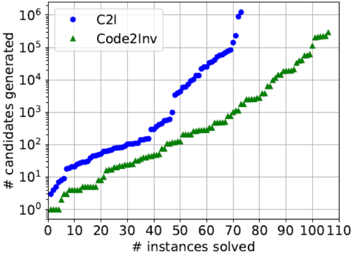
Situations for termination:

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

Experiment Result



(a) verification cost by each solver



(b) sample complexity of C2I and CODE2INV