Lemmaless Induction in Trace Logic

Ahmed Bhayat along with

Pamina Georgiou, Clemens Eisenhofer, Laura Kovács & Giles Reger

SCorCH

Motivation

- Many different program verification techniques such as k-induction, BMC, predicate abstraction etc.
- Most based on SMT / SAT.
- SMT / SAT -based methods can struggle with unbounded loops.
- We provide a method of encoding programs into first-order logic with quantification.
- We introduce induction techniques, suitable for first-order provers, that can provide useful loop invariants.

First-Order Logic

- We work with standard first-order logic with built-in equality ($\simeq$)
- A literal is either an atom A or its negation $\neg A$
- A clause is a disjunction of literals $L_1 \vee L_2 \vee \cdots \vee L_n$
- By $F[t]$ we represent a term t surrounded by a context F

Trace Logic

- Trace logic is an instance of many-sorted first-order logic with theories for natural numbers, integers and timepoints
- Trace logic is useful for stating the semantics of procedural programs
- The signature of trace logic is:

$$\Sigma(\mathcal{L}) = S_{\mathbb{N}} \cup S_{\mathbb{I}} \cup S_{\mathbb{L}} \cup S_{v} \cup S_{n}$$

$S_{\mathbb{N}}$: Signature of the theory of natural numbers

$S_{\mathbb{I}}$: Signature of the theory of Integers

$S_{\mathbb{L}}$: Set of location symbols

S_{v}: Symbols representing program variables

S_{n}: Symbols of sort $\mathbb{N}$ representing the number of times a loop iterates

Running Example

```
Int a[];
const Int len;
Int j = 0;
while(j < len){
 a[j] = 0;
 j = j + 1;
}
```

$\forall pos_{\mathbb{I}}.\, 0 \leq pos \land pos < len \Rightarrow a(end, pos) \simeq 0$

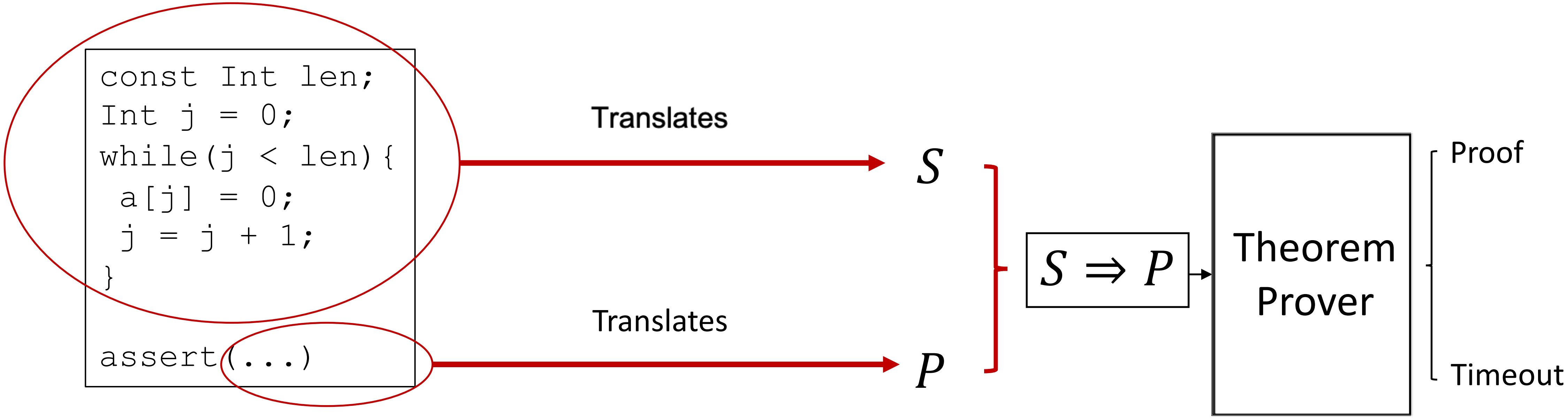
Note on Location Symbols

- The location sort $\mathbb{L}$ is an uninterpreted sort.
- Intuitively, a term $t : \mathbb{L}$ represents a line in a program, e.g., $l3 : \mathbb{L}$ could represent the 3rd line of the running example.
- A line that occurs within a loop can be visited multiple times, so location symbols representing such lines are of type $\mathbb{N} \rightarrow \mathbb{L}$.
- For example, $l6(1) : \mathbb{L}$ represents line 6, at the first iteration of the loop.

The Rapid Verification Tool

- Translates the semantics of a program into trace logic.
- Attempts to prove $S \Rightarrow P$ where S is the program semantics and P some property to be proved.
- The property P can be an arbitrary formula of trace logic and can contain quantifier alternations.
- Currently, Rapid handles a restricted programming language $\mathcal{W}$.

The Rapid Verification Tool

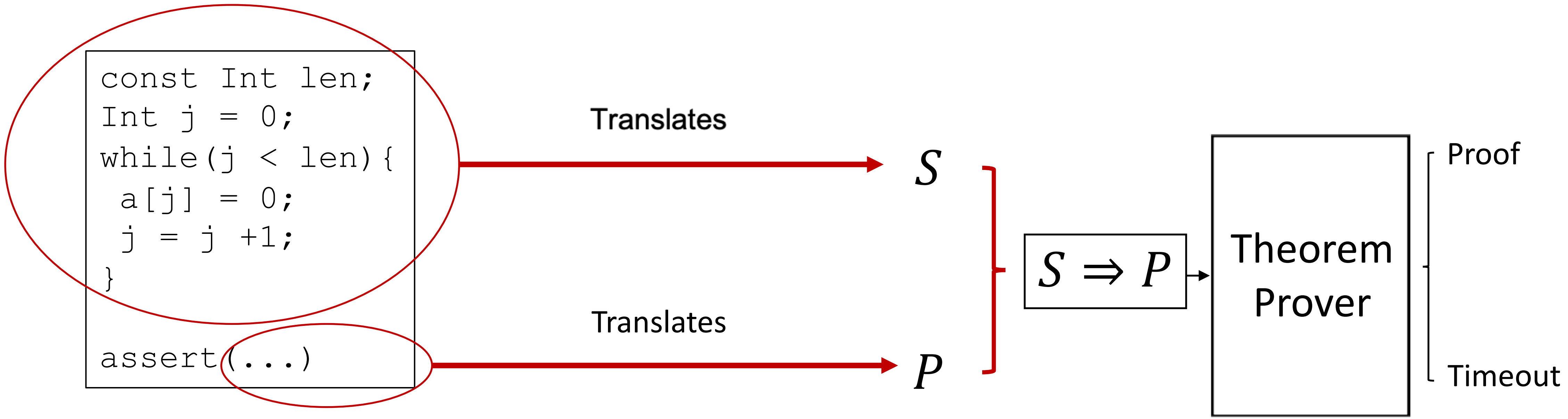


The Vampire Theorem Prover

- Rapid can integrate with any theorem prover capable of reasoning about full first-order logic.
- Currently it integrates with Vampire a theorem prover for first- and higher-order logic based on superposition.
- Vampire supports reasoning in various theories, but not the theory of bit vectors.
- Program integers treated as ideal integers in the logic.

Some of Vampire's trophies

The Rapid Verification Tool



Translation

Expressions

```
Int a
Int b[]
a + c
a < d
```

Assignments

```
a = 10;
a = a + 1;
b[5] = 11;
```

If Statements

```
if(a < 10){
  a = a - 5;
} else {
  a = a + 5;
}
```

While Loops

```
while(x < 10){
  x = x * 2;
}
```

Expressions

- Integer program variables are treated as logic functions from locations to integers.
- For example, an integer variable `a` is translated to a function $a : \mathbb{L} \to \mathbb{I}$.
- Integer array variables are translated with an extra argument: $b : \mathbb{L} \times \mathbb{I} \to \mathbb{I}$.
- Integer and Boolean expressions translated inductively:

`5: a + b`	translates to:	$a(l5) + b(l5)$
`6: (a[5] < 10)`	translates to:	$a(l6, 5) < 10$

Assignments

Assignments can be translated quite simply:

```
5: a = a + 2
```

Translates to:

$$a(l6) \simeq a(l5) + 2 \ \wedge_{v \in S_v \setminus \{a\}} \ v(l6) \simeq v(l5)$$

If Statements

If statements are translated using implications:

```
4: if(a < 10){
5:     a = a - 5;
6: } else {
7:     a = a + 5;
8: }
9:
```

Translates to:

$$a(l4) < 10 \Rightarrow a(l9) \simeq a(l5) - 5 \wedge \neg(a(l4) < 10) \Rightarrow a(l9) \simeq a(l7) + 5$$

While Loops

For loops, we assume termination:

```
5: while(j < len){
6:  a[j] = 0;
7:  j = j + 1;
8: }
9:
```

Translates to:

$$
\begin{aligned}
&\forall it_{\mathbb{N}}.\, it < nl5 \Rightarrow j\big(l5(it)\big) < len \quad \land \\
&\neg\Big(j\big(l5(nl5)\big) < len\Big) \quad \land \\
&\forall pos_{\mathbb{I}}.\, a(l9, pos) \simeq a(l5(nl5), pos) \quad \land \\
&\forall it_{\mathbb{N}}.\, it < nl5 \Rightarrow j\big(l5(it+1)\big) \simeq j\big(l7(it)\big) + 1
\end{aligned}
$$

Where $nl5 : \mathbb{N} \in S_n$

Difficulty with While Loops

- In many cases, loop semantics are not strong enough to prove anything of interest.
- Semantics require strengthening with loop invariants.
- Can add generic invariant schemas (previous work).
- Can introduce dedicated induction inference rules into the solver (this work).

Induction on Loop Counters

Let $p = F[l10(t_{\mathbb{N}})]$ be a formula. To show that p is an invariant of a loop occurring on line 10 of a program, we need to show

$$\begin{aligned} &base\ case: && F[l10(0)] \\ &step\ case: && \forall it_{\mathbb{N}}.\ it < nl10 \wedge F[l10(it)] \Rightarrow F[l10(it+1)] \end{aligned}$$

Allowing us to conclude:

$$\forall it_{\mathbb{N}}.\ it \leq nl10 \Rightarrow F[l10(it)]$$

Finding Invariants

A significant challenge is finding suitable invariants. One possibility is to use the assertions themselves to guide invariant generation.

Some difficulties:

- Assertions may need to be rewritten before they can be useful.
- Within a superposition-based prover, assertions may be split into many clauses.

Finding Invariants

```
1: Int a[];
2: const Int len;
3: Int j = 0;
5: while(j < len){
6:   a[j] = 0;
7:   j = j + 1;
8: }
```

$$\forall pos_{\mathbb{I}}.\ 0 \leq pos \land pos < len \Rightarrow a(end, pos) \simeq 0$$

CNF

$$C_1 = 0 \leq sk \qquad C_2 = sk < len$$
$$C_3 = \neg(a(end, sk) \simeq 0)$$

Rewriting

$$C_1 = 0 \leq sk \qquad C_4 = sk < j(l5(nl5))$$
$$C_5 = \neg(a(l5(nl5), sk) \simeq 0)$$

Induction

$$\neg(C_4[0] \land C_5[0]) \quad \land$$
$$(\forall it.\ it < nl5 \land \neg(C_4[it] \land C_5[it]) \Rightarrow$$
$$\neg(C_4[it+1] \land C_5[it+1]) \quad \Rightarrow$$
$$\forall it.\ it \leq nl5 \Rightarrow \neg(C_4[it] \land C_5[it])$$

$C_1 = 0 \leq sk$

$C_4 = sk < j(l5(nl5))$

$C_5 = \neg(a(l5(nl5), sk) \simeq 0)$

Finding Invariants

$$base\ case:\ \ (sk \geq j(l5(0)) \vee a(l5(0), sk) \simeq 0$$

$$step\ case:\ \ \forall it.\, it < nl5 \wedge (sk \geq j(l5(it)) \vee a(l5(it), sk) \simeq 0) \Rightarrow (sk \geq j(l5(it+1)) \vee a(l5(it+1), sk) \simeq 0)$$

$$conclusion:\ \ \forall it.\, it \leq nl5 \Rightarrow (sk \geq j(l5(it)) \vee a(l5(it), sk) \simeq 0)$$

CNF

$$it > nl5 \vee sk \geq j(l5(it)) \vee a(l5(it), sk) \simeq 0$$

$$\neg(a(l5(nl5), sk) \simeq 0)$$

$$nl5 > nl5 \vee sk \geq j(l5(nl5))$$

$$sk < j(l5(nl5))$$

$$nl5 > nl5$$

Multi-clause Goal Induction

$$\frac{C_1[nl_{\mathtt{w}}] \qquad C_2[nl_{\mathtt{w}}] \quad \ldots \quad C_n[nl_{\mathtt{w}}]}{\mathrm{CNF}\left(\begin{pmatrix} \neg(C_1[0] \wedge C_2[0] \wedge \ldots \wedge C_n[0]) \wedge \\ \forall it_{\mathbb{N}}. \begin{pmatrix} ((it < nl_{\mathtt{w}}) \wedge \neg(C_1[it] \wedge C_2[it] \wedge \ldots \wedge C_n[it])) \rightarrow \\ \neg(C_1[\mathbf{suc}(it)] \wedge C_2[\mathbf{suc}(it)] \wedge \ldots \wedge C_n[\mathbf{suc}(it)]) \end{pmatrix} \end{pmatrix} \\ \rightarrow (\forall it_{\mathbb{N}}. (it < nl_{\mathtt{w}}) \rightarrow \neg(C_1[it] \wedge C_2[it] \wedge \ldots \wedge C_n[it]))\right)}$$

Where $C_1 \cdots C_n$ are all derived from the negated clausified conjecture.

Array Mapping Induction

- Sometimes induction based on safety condition isn’t sufficient.
- This is particularly the case for benchmarks involving multiple loops where we commonly require some form of *forward* reasoning.
- We introduce a separate induction rule called array mapping induction.

Results

- We tested our method on 111 benchmarks coming from the SVCOMP library.
- The verification conditions are custom involved existential and universal quantifiers.
- We compared against a previous version of Rapid, and SeaHorn and Vajra tools

Vampire*	Vampire	SeaHorn	Vajra
93	78	13	47

Future Directions

- Integrate more sophisticated invariant generation procedures into Vampire.
- Extend Rapid framework to reason about pointers and aliasing.
- Extend Rapid framework to parse and reason about c / c++ code.