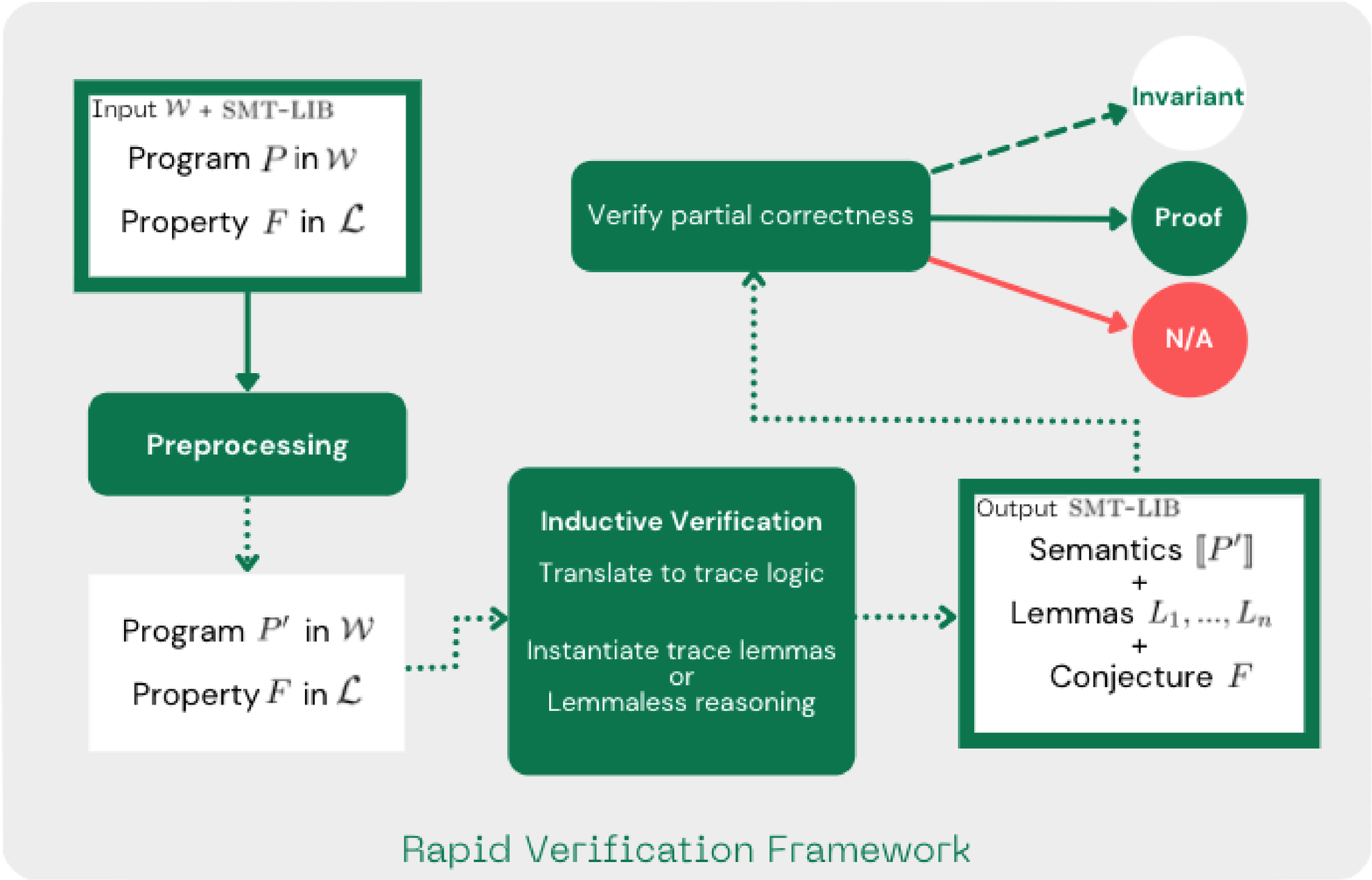
The Rapid Verification Framework

Pamina Georgiou, Ahmed Bhayat, Michael Rawson, Giles Reger, Laura Kovács
Automated Program Reasoning
TU Wien
University of Manchester



Rapid Verification Framework

What are we solving?

```
1    func main() {
2     const Int[] a;
3     Int[] b, c;
4     Int i, j, k = 0;
5     while (i < a.length) {
6      if (a[i] ≥ 0) {
7       b[blength] = a[i];
8       blength++;
9      } else {
10      c[clength] = a[i];
11      clength++;
12     }
13     i = i + 1;
14    }
15   }
16
```

Functional correctness:

- $P := \forall pos_{\mathbb{I}}.\exists pos'_{\mathbb{I}}.((0 \leq pos < blength \wedge a.length \geq 0) \rightarrow 0 \leq pos' < a.length \wedge b(pos) = a(pos'))$

b is initialized by elements of a

Trace Logic Basics

- **Full first-order logic** with equality (over UFDTLIA)
- Program values: standard theory of integers
- Loop iterations: theory of natural numbers $(0, s, p, <)$ (no arithmetic!)
- Reasoning over **timepoints**
  - allows to express induction directly in the language
  - reason about properties of *all loop iterations*
  - reason about the *existence of certain loop iterations*

Timepoints

```
 1  func main() {
 2    const Int[] a;
 3    Int[] b;
 4    Int i = 0;
 5    Int j = 0;
 6    while (i < a.length) {
 7     if (a[i] ≥ 0) {
 8      b[j] = a[i];
 9      j = j + 1;
10     }
11     i = i + 1;
12    }
13  }
14
```

$$l_4 : \mathit{Timepoint}$$

$$l_6(0) : \mathit{Nat} \mapsto \mathit{Timepoint}$$

$$l_6(s(0))$$

$$l_6(n_6)$$

$$l_6(\mathit{it})$$

Program Variables

```
1   func main() {
2     const Int[] a;
3     Int[] b;
4     Int i = 0;
5     Int j = 0;
6     while (i < a.length) {
7      if (a[i] ≥ 0) {
8       b[j] = a[i];
9       j = j + 1;
10     }
11     i = i + 1;
12    }
13  }
14
```

$$i(l_6(0)) \qquad j(l_6(n_6))$$

$$a(i(l_6(0))) \qquad b(l_6(it), j(l_6(it)))$$

$$i : Timepoint \mapsto Int$$

$$a : Int \mapsto Int$$

$$b : Timepoint \times Int \mapsto Int$$

Semantics in Trace Logic

```
1   func main() {
2     const Int[] a;
3     Int[] b;
4     Int i = 0;
5     Int j = 0;
6     while (i < a.length) {
7      if (a[i] ≥ 0) {
8       b[j] = a[i];
9       j = j + 1;
10     }
11     i = i + 1;
12    }
13  }
14
```

$$i(l_4) \simeq 0$$

$$\forall it_{\mathbb{N}} . (it < n_6 \rightarrow$$
$$i(l_6(s(it))) \simeq i(l_{11}(it)) + 1)$$

$$\forall it_{\mathbb{N}} . \Big(it < n_6 \rightarrow$$
$$\big(a(i(l_6(it)) \geq 0) \rightarrow$$
$$b(l_9(it), j(l_6(it))) \simeq a(i(l_6(it))) \big) \Big)$$

What about induction?

Trace Lemma Reasoning

Lemmaless Induction

Trace Lemma Reasoning

- valid formulas, derivable from *instances of the induction scheme*
- *manually identified* set of useful lemmas
- can include *quantifier alternations*
- can include *quantification over loop iterations and variable values*
- can’t be automatically generated by state-of-the-art techniques

Trace Lemma Example

```
1   func main() {
2    const Int[] a;
3    Int[] b;
4    Int i = 0;
5    Int j = 0;
6    while (i < a.length) {
7     if (a[i] ≥ 0) {
8      b[j] = a[i];
9      j = j + 1;
10    }
11    i = i + 1;
12   }
13  }
14
```

`b[k]` remains unchanged after being set

Value Evolution Theorem

Apply bounded induction over equality of variable values

$$\forall p^{\mathbb{I}}.\ \forall it_L^{\mathbb{N}}.\ \forall it_R^{\mathbb{N}}.\ \Big(\Big(\forall it^{\mathbb{N}}.\big(it_L \le it < it_R \wedge b(l_6(it_L),p) = b(l_6(it),p)\big) \rightarrow b(l_6(it_L),p) = b(l_6(s(it)),p)\Big) \rightarrow \big(it_L \le it_R \rightarrow b(l_6(it_L),p) = b(l_6(it_R),p)\big)\Big)$$

Lemmaless Induction

- inbuilt inductive inference rules in first-order theorem proving
- specialized for trace logic
- uses clauses that contain interesting timepoints
- backward reasoning: multi-clause goal induction
- forward reasoning: array-mapping induction

Multi-clause Goal Induction

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

- clauses are **derived from the assertion**
- works well for **properties that are structurally close to the required invariant**, i.e. "almost inductive"

Array-Mapping Induction

- clauses are **derived from program semantics**
- necessary when the required invariant(s) depend more on program behavior (than purely property related)
- **consecutive loops!**

Results

TABLE I: Experimental Results

Total	RAPID$_{\text{std}}$	RAPID$_{\text{lemmaless}}$	DIFFY	SEAHORN
140	91 (5)	103 (10)	61 (1)	17 (0)

What else?

- **Invariant generation** mode: consequence finding from program semantics and trace lemmas
- Current work: extending trace logic for (mutually) **recursive functions/function calls**

If you can think of interesting applications and extensions where quantification is required, come talk to me :)