The Rapid Software Verification Framework

Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, Giles Reger

Automated Program Reasoning Group, TU Wien

October 20, 2022

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[j] = a[i];
 8      j++;
 9     } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

Partial correctness:

b is initialized by elements of a

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[j] = a[i];
 8      j++;
 9     } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

Partial correctness:

b is initialized by elements of a

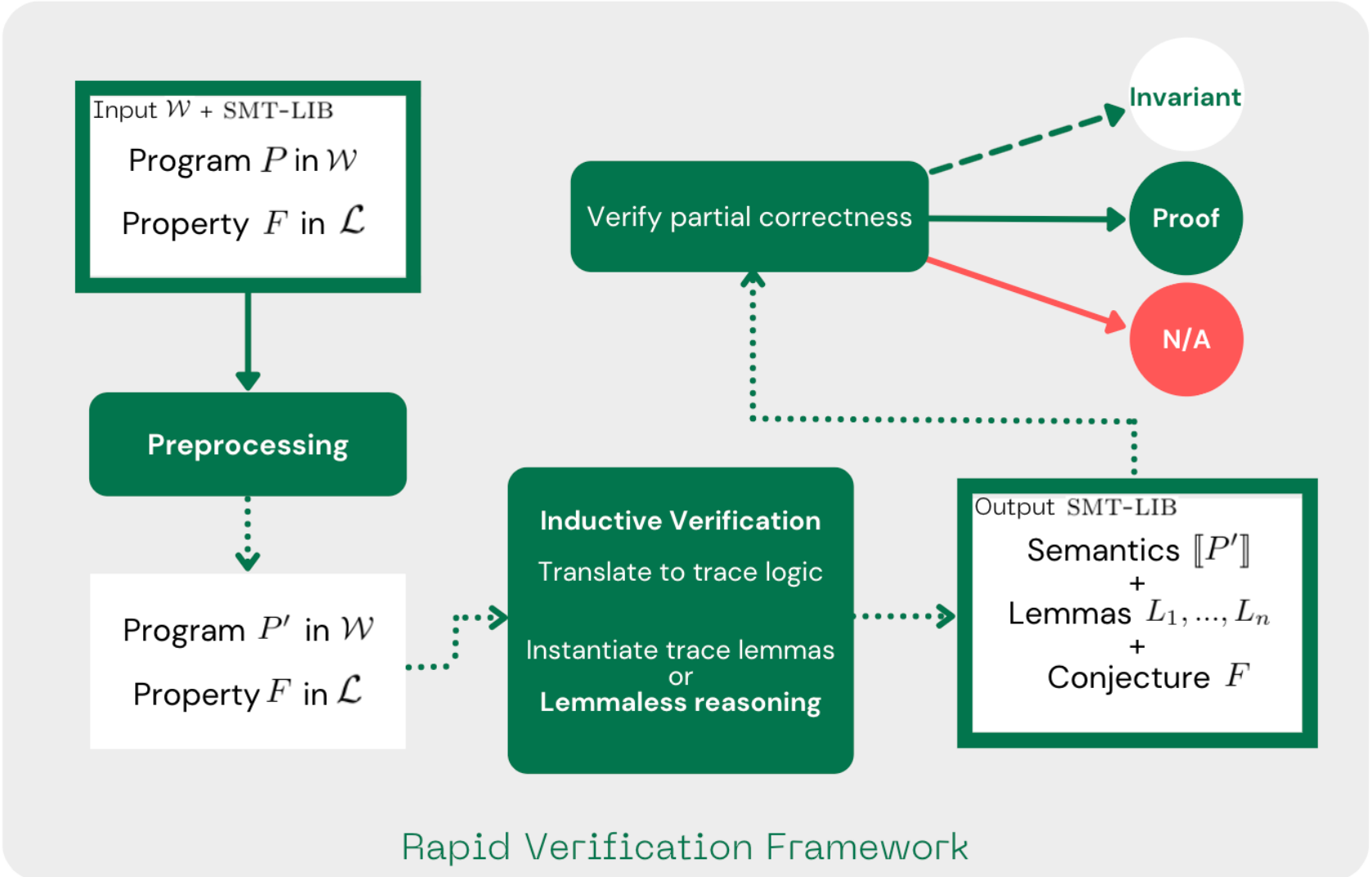
$$\forall pos_{\mathbb{I}}.\exists pos'_{\mathbb{I}}.(0 \leq pos < j \wedge a.length \geq 0 \rightarrow 0 \leq pos' < a.length \wedge b(pos) = a(pos'))$$

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[j] = a[i];
 8      j++;
 9     } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

- Partial correctness
- Programs containing integer and integer-array variables
- Arbitrary amount of loops
- Arbitrarily quantified program properties
- Automated first-order theorem proving

The Rapid Verification Framework



Trace Logic in a Nutshell

- **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*

Semantics based on Trace Logic

```
 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[j] = a[i];
 8      j++;
 9     } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

Conjunction of all statements

$$i(l_4) \simeq 0$$

$$\forall it_{\mathbb{N}} \cdot \Big(it < n_5 \rightarrow$$

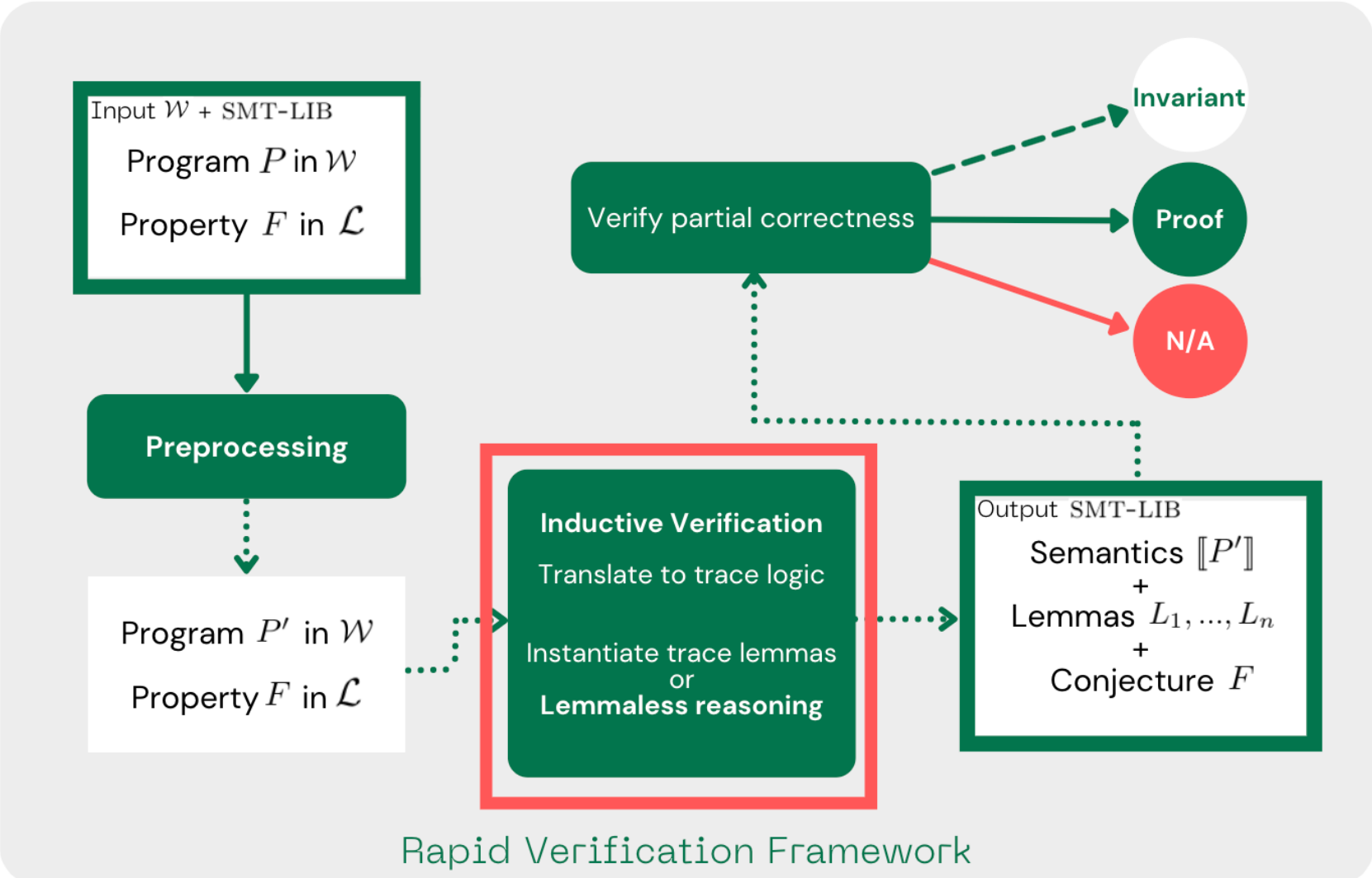
$$i(l_5(s(it))) \simeq i(l_{13}(it)) + 1$$

$$\wedge$$

$$\Big(a(i(l_5(it)) \geq 0) \rightarrow$$

$$b(l_9(it), j(l_5(it))) \simeq a(i(l_5(it))) \Big) \Big)$$

What about induction?



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

Lemmaless Induction

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

Multi-clause Goal Induction

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

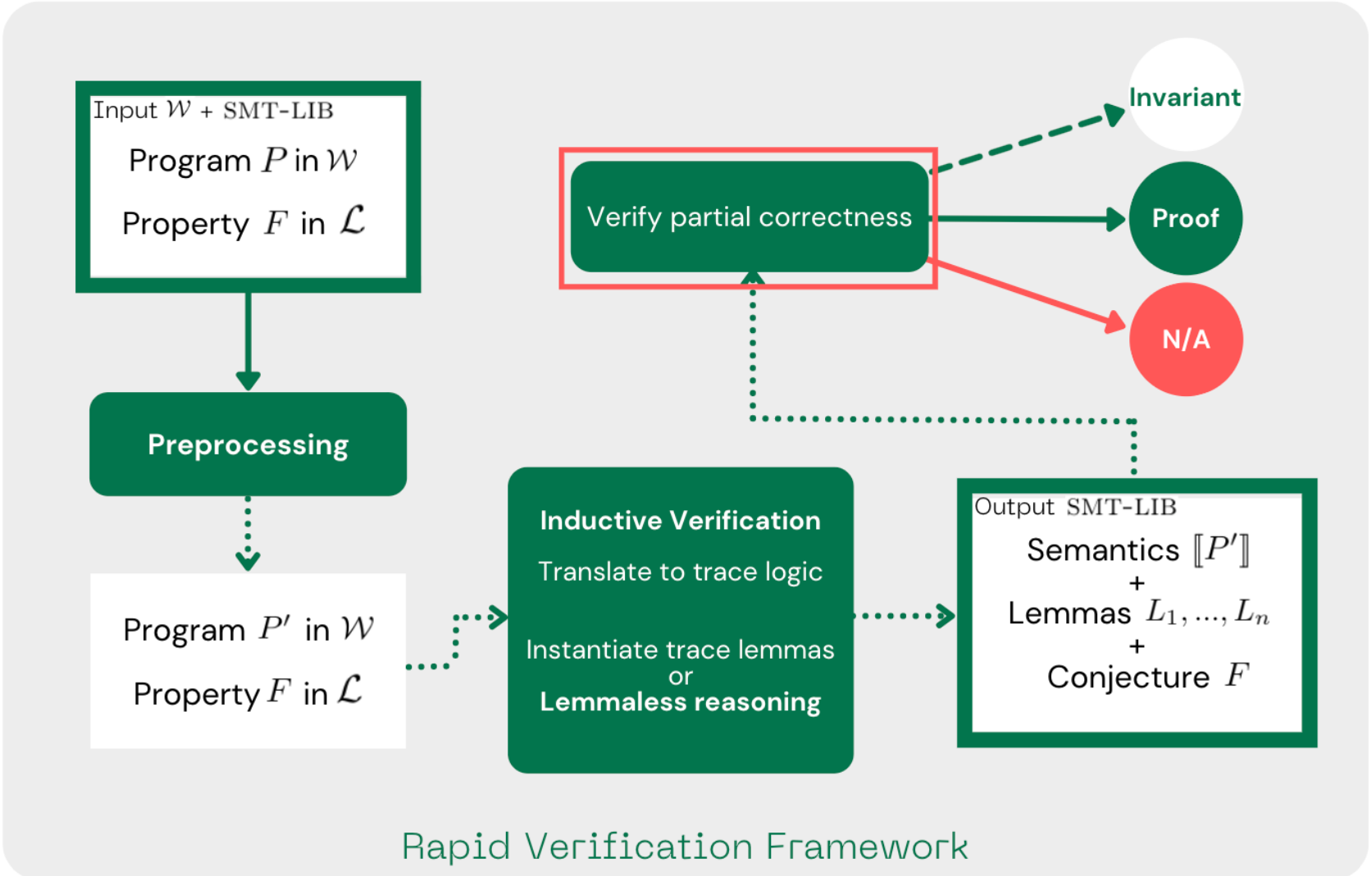
- clauses are *derived from safety assertion*
- works well for *properties that are structurally close to the required invariant*

Array-mapping Induction

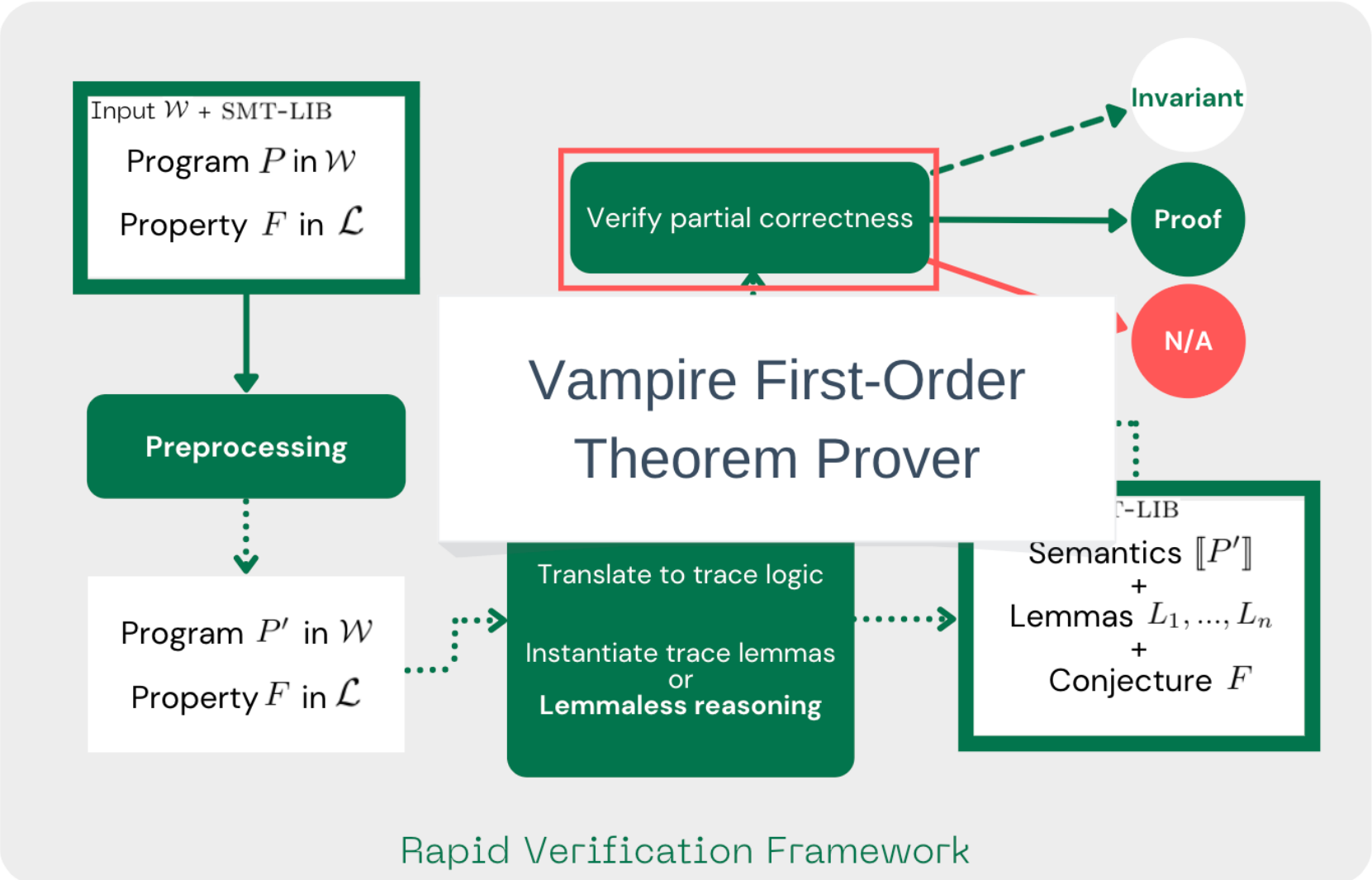
```
 1  func main(){
 2   Int[] a;
 3   Int i, j = 0;
 4   const Int n;
 5   while(i < a.length) {
 6    a[i] = a[i] + n;
 7    i = i + 1;
 8   }
 9   while(j < a.length) {
10    a[j] = a[j] - n;
11    j = j + 1;
12   }
13  }
14
```

- clauses are *derived from program semantics*
- necessary when required invariant(s) depend on program behavior rather than safety assertion
- *consecutive loops!*

How to discharge verification conditions?



How to discharge verification conditions?



Results

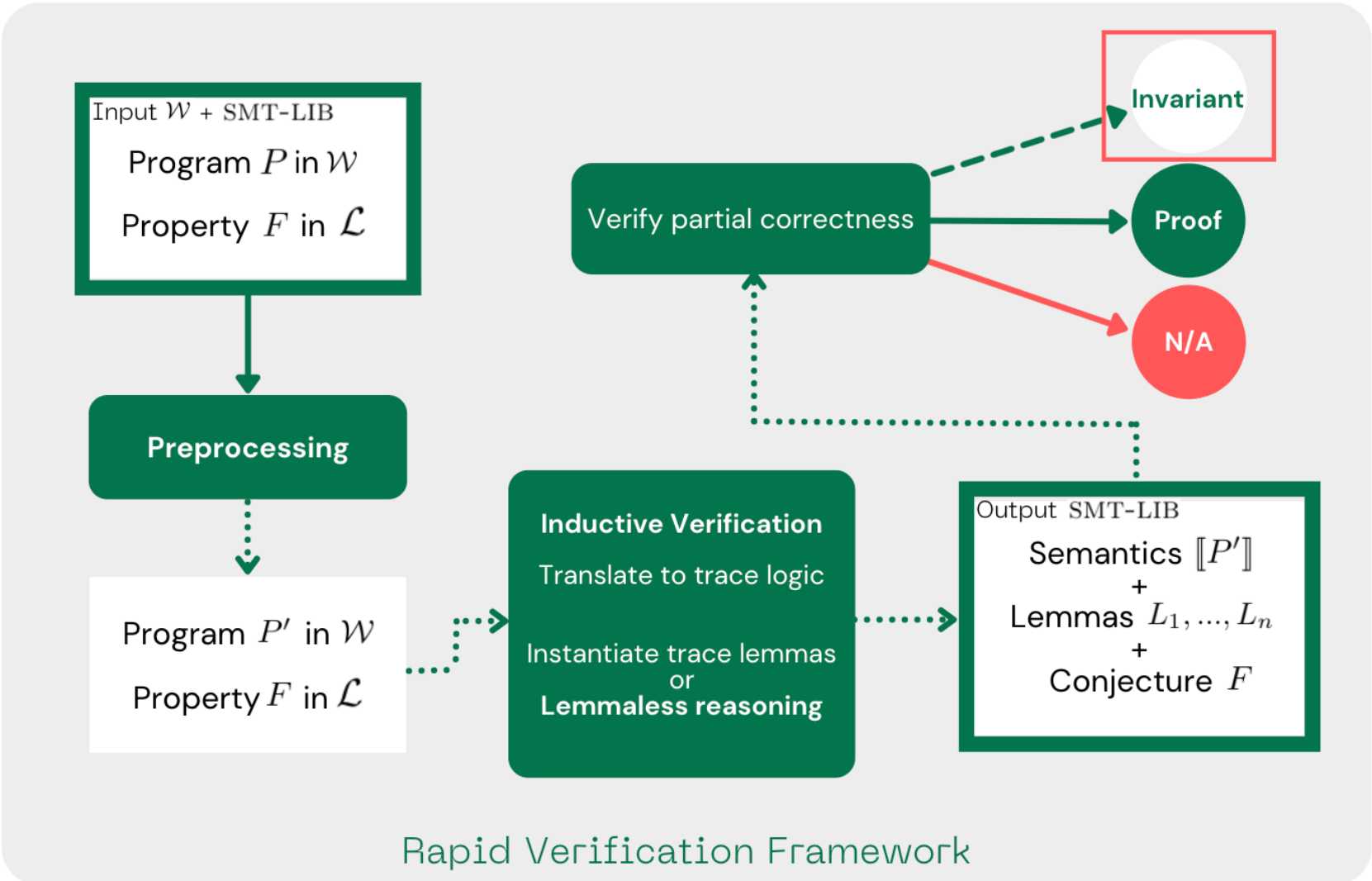
TABLE I: Experimental Results

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

Fin

Thank you for your attention!

What about invariants?



Invariant Generation

- *consequence finding* from semantics and trace lemmas until a conjunction of clauses proves a postcondition
- allows integration with other tools
- works (at most) for already found proofs

Semantics based on Trace Logic

```
 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[j] = a[i];
 8      j++;
 9     } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

Timepoint reasoning

$$l_4 : \mathit{Timepoint}$$

$$l_5(0) : \mathit{Nat} \mapsto \mathit{Timepoint}$$

$$l_5(s(0))$$

$$l_5(n_5)$$

$$l_5(it)$$

Semantics based on Trace Logic

```
 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[j] = a[i];
 8      j++;
 9     } else {
10      c[k] = a[i];
11      k++;
12     }
13     i++;
14    }
15   }
16
```

Program variables

$i(l_5(0))$ $\quad$ $j(l_5(n_5))$

$a(i(l_5(0)))$ $\quad$ $b(l_5(it), j(l_5(it)))$

$$i : Timepoint \mapsto Int$$

$$a : Int \mapsto Int$$

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