netidee STIPENDIEN

Automated Software Verification with First-Order Theorem Provers

Endbericht | Call 16 | Stipendium ID 5761

Inhalt

1. Einführung

Software verification is ubiquitous in computing. While automating verification in undecidable fragments is hard, it is a worthwhile goal to establish safe security-critical virtual infrastructures. *This work investigates and promotes the use of full first-order theorem provers for software verification for programs containing (recursive) data structures and (recursive) function calls.*

Automated software verification is a realm mainly depending on satisfiability modulo theories (SMT)-solvers, thus limiting the logical representation of programming language semantics as well as safety properties to the expressiveness of such solvers. However, most automated verification approaches depend on SMT-solvers to discharge verification conditions. Consequently they are mostly restricted to universal quantification, rarely some existential quantification, but almost no quantifier alternations. Automated first-order theorem provers, in contrast, are not limited in the use of quantification and can work with formulas of universal, existential but also alternating quantification which promotes alternative ways of automated verification. While winning many competitions in first-order reasoning, they are rarely promoted for verification purposes.

In this dissertation, I'm aiming to make first-order theorem proving more applicable to automated program reasoning and verification. Specifically, we want to improve the *automation of inductive reasoning over loops and recursive functions with first-order theorem provers*. The integral part of this work is automating inductive reasoning which is famously hard as verifying the correctness of any formula containing un- or semi-decidable theories such as integer arithmetic is an semi-decidable problem in mathematics. Thus, mathematically speaking verification heavily depends upon finding either decidable fragments of logic, or, as we do in this work, find ways to make our approaches applicable to a wide set of problems even in the sight of undecidability of the underlying logical framework.
We achieve this goal through multiple works expanding on inductive reasoning for trace logic, invariant generation as well as investigating functional programming paradigms and provide a proof of concept of proving functional sorting algorithms correct.

We already introduced our verification framework Rapid based on expressing semantics in *trace logic,* a many-sorted first-order logic with built-in equality, integers and natural numbers. Our original efforts have been relying on manually identified lemmas - coined *trace lemmas* - that are applicable to a wide range of programs containing loops and unbounded data structures as arrays over integer arithmetic. However, these are limited to certain types of reasoning and are not always enough

to find a proof. Additionally instantiating trace lemmas for a growing number of program variables results in many unnecessary lemmas added due to automation, thus making the search space for a proof bigger and bigger.
A massive improvement was achieved by relying less on trace lemmas for inductive reasoning but leveraging inductive reasoning directly in the underlying first-order theorem prover Vampire. To this end, we exploited inductive inferences in the first-order superposition calculus for reasoning about loops in trace logic. That is, we adopted built-in induction schemes for bounded induction over loop iterations in the underlying calculus, thus verifying inductive reasoning steps during proof search. Our results show that such a method of inductive reasoning is superior to our prior works using trace lemmas.
Beyond trace logic, we also tackle functional programming paradigms with built-in induction: our most recent efforts focus on inductive/functional semantics of algorithms transforming lists of multiple types, that is we do not only prove program correctness for lists of integers or naturals, but we can handle programs containing lists of any data type that provides a partial order. This is powered by type parameterisation for list data types. Additionally, we formalize the first-order semantics of functional programs and provide a proof of concept of verifying partial correctness with computation induction on functional sorting algorithms. More details on recent developments are summarized below.

2. Allgemeines

The core of my PhD thesis is to make first-order theorem proving based on the superposition inference calculus more usable for software verification purposes. Specifically, we investigate how we can automate inductive reasoning that is necessary for verification with regards to program loops or recursive function calls.

In the first part of my thesis, we adopted *trace logic* for reasoning in full first-order logic with built-in equalities and theories. We adapted the inductive reasoning to *lemmaless reasoning* built-in into the automated theorem prover, thus increasing the degree of automation even more and making trace logic more applicable to programs that contain (consecutive, as well as nested) loops.

While the original goal was to extend trace logic and its first-order semantics to function calls, we realized that making use of lemmaless reasoning through inductive inferences in the underlying prover is a more fruitful mission on the long run. That is, adapting inductive inferences to reason about functional programs that contain inductive function definitions and potentially many recursive calls, including mutual recursion, is the more powerful solution. Moreover, we also came to the conclusion that while trace logic is a powerful tool for reasoning about loops due to this inherent notion of time, it is less so for functional algorithms. Extending the semantics in trace logic to function calls thus proved to be too convoluted and simply too big for the prover to effectively pursue proof search in reasonable time. Moreover, we realised that a purely functional programming paradigm leads to very simple program semantics in first-order logic when using inductive definitions.

In our latest work, we thus adapted a first-order semantics for functional programs and showed how structural and computation induction schemes can be adapted for inductive inferences during proof search, specifically during saturation. We showcased how we can reason about such programs by verifying a purely functional version of the famous QuickSort algorithm, among other sorting algorithms, for this work.
The crux of this work lies in compositional reasoning: rather than relying on a priori (user-provided) inductive annotations such as invariants, we split the proof into multiple proof obligations whenever necessary. This is the case when the reasoner can only find induction axioms that are too strong to be proven correct within a certain time limit. However each of these proof splits is established, i.e. lemma correctness is verified, in the underlying automated theorem prover with built-in induction. In contrast to most verification tools, we automate the inductive reasoning rather than just assuming it.

3. Ergebnisse

In the course of the netidee stipend, three scientific publications were created of which one is still under peer review. All of the works below further the use of first-order theorem provers to showcase its efficiency and possibilities of usage when it comes to automated software verification. They are listed below:

(1) Bhayat, A., Georgiou, P., Eisenhofer, C., Kovács, L., & Reger, G. (2022, September). Lemmaless induction in trace logic. In *International Conference on Intelligent Computer Mathematics—CICM 2022* (pp. 191-208). Cham: Springer International Publishing.

(2) Georgiou, P., Gleiss, B., Bhayat, A., Rawson, M., Kovács, L., & Reger, G. (2022, October). The RAPID Software Verification Framework. In *Conference on Formal Methods in Computer-aided Design—FMCAD 2022* (p. 255).

(3) Georgiou, P., Hajdu, M., Kovács, L. (2023, July). Sorting without Sorts. In *EasyChair Preprint no. 10632—EasyChair 2023* (currently under review at the 25th International Conference on Verification, Model Checking, and Abstract Interpretation—VMCAI 2024)

Specifically (1) shows how in-built inductive inference rules can be adjusted towards trace logic for lemmaless reasoning to verify complex properties of programs containing loops and arrays that may contain quantifier alternations fully automatically with first-order theorem proving.

(2) summarizes the capabilities of the underlying tool RAPID that allows for the automated translation of while-language programs and their properties into first-order reasoning problems in trace logic while performing lightweight static analysis to equip the reasoning problems with further information for the prover in the form of trace lemmas. Additionally, we introduce symbol elimination for trace logic to provide a further use case and enable invariant generation with first-order theorem provers.

In (3) we investigate in-built inductive inferences in first-order theorem provers further and highlight its capabilities by proving (among others) the functional version of the quick sort algorithm correct, even in the sight of recursive data structures and recursive function calls - an algorithm that is notoriously known to be hard to be proved correct in an automated way. To the best of our knowledge, no fully automated proof of this algorithm exists which emphasises the impact of our method presented in this paper. Moreover, this work shows that first-order theorem proving can be used to efficiently find necessary induction axioms automatically in practice. The paper is currently under review for publication at FASE 2024.

All three publications are attached to this final report.

4. Geplante weiterführende Aktivitäten

In terms of my PhD thesis, my contributions towards automating induction in first-order theorem provers for software verification will be summarised in a coherent thesis amplifying the motivation for the work during and prior to my netidee scholarship.
During the writing process, I will also review prior works on verifying security relevant properties, known as hyper-properties, and evaluate our current state of work on these benchmarks.
Lastly, I'm planning to defend my thesis in March 2024.

5. Anregungen für Weiterführung durch Dritte

My thesis will show that first-order theorem (FOL) proving is a widely underrated tool when it comes to software verification. There is a lot of opportunity still to be exploited towards automating the verification process even more.

An immediate continuation of the work lies in building a framework for functional programming languages/algorithms based on our most recent work, similar to the

RAPID framework for while-based languages. Another question is how we can automate the proof splits necessary in our latest work based on some heuristics.

A more general open question in automated theorem proving is how to find counterexamples. One of the major drawbacks of using FOL provers is that they mostly check validity of formulae but, due to semi-decidability of first-order logic with (undecidable) theories, rarely offer the possibility to produce counterexamples to validity. They simply diverge and run until the process times out. Thus, counterexample/model finding in FOL proving would be a major step to look into when it comes to using FOL provers for more large-scale software verification processes. While there are certainly many open questions that can be investigated, this is in my opinion the most prevalent one to make automated theorem proving more convincing for verification purposes. There is prior research on this subject in the paramodulation community, however, none of them are particularly investigating software correctness counterexample finding in such calculi.

Lemmaless Induction in Trace Logic

Ahmed Bhayat[1(✉)], Pamina Georgiou[2], Clemens Eisenhofer[2], Laura Kovács[2], and Giles Reger[1]

[1] University of Manchester, Manchester, UK
{ahmed.bhayat,giles.reger}@manchester.ac.uk
[2] TU Wien, Vienna, Austria
{pamina.georgiou,clemens.eisenhofer,laura.kovacs}@tuwien.ac.at

Abstract. We present a novel approach to automate the verification of first-order inductive program properties capturing the partial correctness of imperative program loops with branching, integers and arrays. We rely on trace logic, an instance of first-order logic with theories, to express first-order program semantics by quantifying over program execution timepoints. Program verification in trace logic is translated into a first-order theorem proving problem where, to date, effective reasoning has required the introduction of so-called trace lemmas to establish inductive properties. In this work, we extend trace logic with generic induction schemata over timepoints and loop counters, reducing reliance on trace lemmas. Inferring and proving loop invariants becomes an inductive inference step within superposition-based first-order theorem proving. We implemented our approach in the RAPID framework, using the first-order theorem prover VAMPIRE. Our extensive experimental analysis shows that automating inductive verification in trace logic is an improvement compared to existing approaches.

1 Introduction

Automating the verification of programs containing loops and recursive data structures is an ongoing research effort of growing importance. While different techniques for proving the correctness of such programs are in place [5,6,10,13], most existing tools in this realm are heavily based on *satisfiability modulo theories* (SMT) backends [4,8] that come with strong theory reasoning but have limitations in quantified reasoning. In contrast, first-order theorem provers enable quantified reasoning modulo theories [19,24,25], such as linear integer arithmetic and arrays. First-order reasoning can thus complement the aforementioned verification efforts when it comes to proving program properties with complex quantification, as evidenced in our original work on the RAPID framework [11] which utilised the VAMPIRE theorem prover [2,20].

At a high level, the RAPID framework [11] works by translating a program into *trace logic*, adding a number of ad hoc trace lemmas, asserting a desired property, and then running an automated theorem prover on the result. The effectiveness of this approach depends on the underlying trace lemmas. This

K. Buzzard and T. Kutsia (Eds.): CICM 2022, LNAI 13467, pp. 191–208, 2022.
https://doi.org/10.1007/978-3-031-16681-5_14

paper focuses on building induction support into the Vampire theorem prover to reduce reliance on these lemmas.

To understand the role of these trace lemmas (and therefore, what support must be added to the theorem prover) we briefly overview trace logic and the Rapid framework in a little more detail. Trace logic is an instance of first-order logic with theories, such that the program semantics of imperative programs with loops, branching, integers, and arrays can be directly encoded in trace logic. A key feature of this encoding is tracking program executions by quantifying over execution *timepoints* (rather than only over single states), which may themselves be parameterised by *loop iterations*. In principle, we can check whether a translated program entails the desired property in trace logic using an automated theorem prover for first-order logic. In our case, we make use of the saturation-based theorem prover Vampire which implements the superposition calculus [3]. However, a straightforward use of theorem proving often fails in establishing validity of program properties in trace logic, as the proof requires some specific induction, in general not supported by superposition-based reasoning.

In our previous work [11], we overcame this challenge by introducing so-called *trace lemmas* capturing common patterns of inductive loop properties over arrays and integers. Inductive loop reasoning in trace logic is then achieved by generating and adding trace lemma instances to the translated program. However, there are two significant limitations to using trace lemmas:

1. Trace lemmas capture inductive patterns/templates that need to be manually identified, as induction is not expressible in first-order logic. As such, they cannot be inferred by a first-order reasoner, implying that the effectiveness of trace logic reasoning depends on the expressiveness of manually supplied trace lemmas.
2. When instantiating trace lemmas with appropriate inductive program variables, a large number of inductive properties are generated, causing saturation-based proof search to diverge and fail to find program correctness proofs in reasonable time.

In this paper we address these limitations by reducing the need for trace lemmas. We achieve this by introducing a couple of novel induction inferences. Firstly, *multi-clause goal induction* which applies induction in a goal oriented fashion as many safety program assertions are structurally close to useful loop invariants. Secondly, *array mapping induction* which covers certain cases where the required loop invariant does not stem from the goal. Specifically, we make the following contributions:

Contribution 1. We introduce two new inference rules, *multi-clause goal* and *array mapping* induction, for *lemmaless induction* over loop iterations (Sects. 5–6). The inference rules are compatible with any saturation-based inference system used for first-order theorem proving and work by carrying out induction on terms corresponding to final loop iterations.

```
1   func main() {
2     const Int[] a;
3     const Int[] b;
4     Int[] c;
5     const Int length;
6     Int i = 0;
7
8     while (i < length) {
9       c[2*i] = a[i]
10      c[(2*i) + 1] = b[i]
11      i = i + 1;
12    }
13  }
14  assert (∀pos_𝕀.∃l_𝕀.((0 ≤ pos < (2 × length))
15      → c(main_end, pos) = a(l) ∨ c(main_end, pos) = b(l)
      ))
```

Fig. 1. Copying elements from arrays `a` and `b` to even/odd positions in array `c`.

Contribution 2. We implemented our approach in the first-order theorem prover Vampire [20]. Further, we extended the Rapid framework [11] to support inductive reasoning in the automated backend (Sect. 7). We carry out an extensive evaluation of the new method (Sect. 8) comparing against state-of-the-art approaches SeaHorn [12,13] and Vajra/Diffy [5,6].

2 Motivating Example

We motivate our work with the example program in Fig. 1. The program iterates over two arrays `a` and `b` of arbitrary, but fixed length `length` and copies array elements into a new array `c`. Each even position in `c` contains an element of `a`, while each odd position an element of `b`. Our task is to prove the safety assertion at line 14: at the end of the program, every element in `c` is an element from `a` or `b`. This property involves (i) alternation of quantifiers and (ii) is expressed in the first-order theories of linear integer arithmetic and arrays. Note that in the safety assertion, the program variable `length` is modeled as a logical constant of the same name of sort integer, whilst the constant arrays `a` and `b` are modeled as logical functions from integers to integers. The mutable array variable `c` is additionally equipped with a timepoint argument `main_end`, indicating that the assertion is referring to the value of the variable at the end of program execution.

Proving the correctness of this example program remains challenging for most state-of-the-art approaches, such as [5,6,10,12], mainly due to the complex quantified structure of our assertion. Moreover, it cannot be achieved in the current Rapid framework either, as existing trace lemmas do not relate the values of multiple program variables, notably equality over multiple array variables. In fact, to automatically prove the assertion, we need an inductive property/trace lemma formalizing that each element at an even position in `c` is an element of `a` or `b` at each valid loop iteration, thereby also restricting the bounds of the loop counter variable `i`. Naïvely adding such a trace lemma would be highly inefficient as automated generation of verification conditions would introduce many instances that are not required for the proof.

3 Related Work

Most of recent research in verifying inductive properties of array-manipulating programs focuses on quantified invariant generation and/or is mostly restricted to proving universally quantified program properties. The works [10,13] generate universally quantified inductive invariants by iteratively inferring and strengthening candidate invariants. These methods use SMT solving and as such are restricted to first-order theories with a finite model property. Similar logical restrictions also apply to [23], where linear recurrence solving is used in combination with array-specific proof tactics to prove quantified program properties. A related approach is described in [6], where relational invariants instead of recurrence equations are used to handle universal and quantifier-free inductive properties. Unlike these, our work is not limited to universal invariants but can infer and prove inductive program properties with alternations of quantifiers.

With the use of extended expressions and induction schemata, our work shares some similarity with template-based approaches [16,21,26]. These works infer and prove universal inductive properties based on Craig interpolation, formula slicing and/or SMT generalizations over quantifier-free formulas. Unlike these works, we do not require any assumptions on the syntactic shape of the first-order invariants. Moreover, our invariants are not restricted to the shape of our induction schemata. Rather, we treat inductive (invariant) inferences as additional rules of first-order theorem provers, maintaining thus the efficient handling of arbitrary first-order quantifiers. Our framework can be used in arbitrary first-order theories, even with theories that have no interpolation property and/or a finite axiomatization, as exemplified by our experimental results using inductive reasoning over arrays and integers.

Inductive theorem provers, such as ACL2 [17] and HipSpec [7], implement powerful induction schemata and heuristics. However these provers, to the best of our knowledge, automate inductive reasoning for only universally quantified inductive formulas using a goal/subgoal architecture, for which user-guidance is needed to split conjectures into subgoals. In contrast, our work can prove formulas of full first-order logic by integrating and fully automating induction in saturation-based proof search. By combining induction with saturation, we allow these techniques to interleave and complement each other, something that pure induction provers cannot do. Unlike tools such as Dafny [22], our approach is fully automated requiring no user annotations.

First-order theorem proving has been used to derive invariants with alternations of quantifiers in our previous work [11]. Our current work generalizes the inductive capabilities of [11] by reducing the expert knowledge of [11] in introducing inductive lemmas to guide the process of proving inductive properties.

4 Preliminaries

Many-Sorted First-Order Logic. We consider standard many-sorted first-order logic with built-in equality, denoted by $\simeq$. By $s = F[u]$ we indicate that the term u is a subterm of s surrounded by (a possibly empty) context F.

We use x, y to denote variables, l, r, s, t for terms and sk for Skolem symbols. A *literal* is an atom A or its negation $\neg A$. A *clause* is a disjunction of literals $L_1 \vee ... \vee L_n$, for $n \geq 0$. Given a formula F, we denote by $\text{CNF}(F)$ the clausal normal form of F.

For a logical variable x of sort S we write x_S. A *first-order theory* denotes the set of all valid formulas on a class of first-order structures. Any symbol in the signature of a theory is considered *interpreted*. All other symbols are *uninterpreted*. In particular, we use the theory of linear integer arithmetic denoted by $\mathbb{I}$ and the boolean sort $\mathbb{B}$. We consider natural numbers as the term algebra $\mathbb{N}$ with four symbols in the signature: the constructors `0` and successor `suc`, as well as `pred` and $<$ respectively interpreted as the predecessor function and less-than relation. Note that we do not define any arithmetic on naturals. We assume familiarity with the basics of saturation theorem proving.

4.1 Trace Logic $\mathcal{L}$

Trace logic, denoted as $\mathcal{L}$, is an instance of many-sorted first-order logic with theories. Its signature is $\Sigma(\mathcal{L}) := S_\mathbb{N} \cup S_\mathbb{I} \cup S_\mathbb{L} \cup S_V \cup S_n$, includes respectively the signatures of the theory of natural numbers $\mathbb{N}$ (as a term algebra), the inbuilt integer theory $\mathbb{I}$, a set $S_\mathbb{L}$ of timepoints (also referred to as *locations*), a set of symbols representing program variables S_V, as well as a set of symbols representing last iteration symbols S_n. For more details on trace logic, refer to [11].

4.2 Programming Model $\mathcal{W}$

We consider programs written in a WHILE-like programming language $\mathcal{W}$, as given in the (partial) language grammar of Fig. 2. Programs in $\mathcal{W}$ contain mutable and immutable integer as well as integer-array program variables and consist of a single top-level function `main` comprising arbitrary nestings of while-loops and if-then-else branching. We consider expressions over booleans and integers without side effects.

4.3 Translating Expressions to Trace Logic

Locations and Timepoints. We consider programs as sets of locations over time: given a program statement `s`, we denote its location by l_s of type $\mathbb{L}$, the location/timepoint sort, corresponding to the line of the program where the statement appears. When `s` is a while-loop the corresponding location is revisited at multiple timepoints of the execution. Thus, we model such locations

```
   program ::= function
  function ::= func main(){ subprogram }
subprogram ::= statement | context
   context ::= statement; ... ; statement
 statement ::= atomicStatement
             | if( condition ){ context } else { context }
             | while( condition ){ context }
```

Fig. 2. Grammar of $\mathcal{W}$.

as functions over *loop iterations* $l_s : \mathbb{N} \mapsto \mathbb{L}$, where the argument of sort $\mathbb{N}$ intuitively corresponds to the number of loop iterations. Further, for each loop statement s we model the last loop iteration by a symbol $nl_s \in S_n$ of target sort $\mathbb{N}$. Let p be a program statement or context. We use $start_{\texttt{p}}$ to denote the location at which the execution of p has started and $end_{\texttt{p}}$ to denote the location that occurs just after the execution of p. We use *main_end* to denote the location at the end of the main function.

Example 1. Consider line 6 of our running example in Fig. 1. Term l_6 corresponds to the timepoint of the first assignment of 0 to program variables i while $l_8(\texttt{0})$ and $l_8(nl_8)$ denote the timepoints of the loop at the first and last loop iteration respectively. Further, we can quantify over all executions of the loops by quantifying over all iterations smaller than the last e.g. $\forall it_{\mathbb{N}}.it < nl_8 \rightarrow F[l_8(it)]$ where $F[l_8(it)]$ is some first-order formula.

Program Variables. Program variable are expressed as functions over timepoints. We express an integer variable v as a function $v : \mathbb{L} \mapsto \mathbb{I}$, where $v \in S_V$. Let tp be a term of sort $\mathbb{L}$. Then, $v(tp)$ denotes the value of v at timepoint tp. We model numeric array variables v with an additional argument of sort $\mathbb{I}$ to denote the position of an array access. We obtain $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$. Immutable variables are modelled as per their mutable counterparts, but without the timepoint argument.

Example 2. To denote program variable i at the location of the assignment in line 6, we use the equation $i(l_6) \simeq 0$. For the first assignment of c within the loop, we write $c(l_8(it), 2 \times i(l_8(it))) \simeq a(i(l_8(it)))$ for some iteration it. As a is a constant array, the timepoint argument is omitted.

Program Expressions. Let e be an arbitrary program expression. We write $[\![\texttt{e}]\!](tp)$ to denote the logical denotation of e at timepoint tp. We do not provide the full inductive definition of the denotation function $[\![\;]\!](tp)$ here, just a few of its cases. If e is an integer variable v, then $[\![\texttt{e}]\!](tp) = v(tp)$. If e is an integer array access of the form $\mathtt{v}[\mathtt{e}_1]$, then $[\![\mathtt{e}]\!](tp) = v(tp, [\![\mathtt{e}_1]\!](tp))$. If $\mathtt{e}$ is an expression of the form $\mathtt{e}_1 + \mathtt{e}_2$, then $[\![\mathtt{e}]\!](tp) = [\![\mathtt{e}_1]\!](tp) + [\![\mathtt{e}_2]\!](tp)$.

Common Abbreviations. Let $\mathtt{e}, \mathtt{e}_1, \mathtt{e}_2$ be program expressions, tp_1, tp_2 be two timepoints and $v \in S_V$ denote the functional representation of a program variable. The trace logic formula $v(tp_1) \simeq v(tp_2)$ asserts that the variable v has the same value at timepoints tp_1 and tp_2. We introduce definitions for two formulas that are widely used in defining the axiomatic semantics of $\mathcal{W}$ in the next section. To ease the notational burden, we ignore array variables in the definitions provided. Firstly, we introduce a definition for the formula that expresses that the value of a variable v changes between timepoints tp_1 and tp_2 whilst the values of all other variables remain the same.

$$Update(v, e, tp_1, tp_2) \quad := \quad v(tp_2) \simeq [\![\mathtt{e}]\!](tp_1) \wedge \textstyle\bigwedge_{v' \in S_V \setminus \{v\}} v'(tp_1) \simeq v'(tp_2),$$

Secondly, we introduce a definition for the formula that expresses that the value of all variables stays the same between timepoints tp_1 and tp_2

$$EqAll(tp_1, tp_2) := \bigwedge_{v \in S_V} v(tp_1) \simeq v(tp_2)$$

4.4 Axiomatic Semantics of $\mathcal{W}$ in $\mathcal{L}$

The semantics of a program in $\mathcal{W}$ is given by the conjunction of the respective axiomatic semantics of each program statement of $\mathcal{W}$ occurring in the program. In general, we define reachability of program statements over timepoints rather than program states. We briefly recall the axiomatic semantics of assignments and while-loops respectively, again ignoring the array variable case.

Assignments. Let $\mathtt{s}$ be an assignment `v = e`, where `v` is an integer-valued program variable and `e` is an expression. The evaluation of `s` is performed in one step such that, after the evaluation, the variable `v` has the same value as `e` before the evaluation while all other variables remain unchanged. We obtain

$$[\![\mathtt{s}]\!] := Update(v, e, start_{\mathtt{s}}, end_{\mathtt{s}}) \tag{1}$$

While-Loops. Let `s` be the while-statement `while(Cond){c}` where `Cond` is the *loop condition.* The semantics of `s` is given by the conjunction of the following properties: (2a) the iteration nl_s is the first iteration where `Cond` does not hold anymore, (2b) jumping into the loop body does not change the values of the variables, (2c) the values of the variables at the end of evaluating the loop `s` are equal to the values at the loop condition location in iteration nl_s. As such, we have

$$\begin{aligned}
[\![\mathtt{s}]\!] := \quad & \forall it^s_{\mathbb{N}}.\ (it^s < nl_{\mathtt{s}} \rightarrow [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^s))) \\
\wedge \quad & \neg[\![\mathtt{Cond}]\!](tp(nl_{\mathtt{s}})) && \text{(2a)} \\
\wedge \quad & \forall it_{\mathbb{N}}.\ (it < nl_{\mathtt{s}} \rightarrow EqAll(start_{\mathtt{c}}, tp_{\mathtt{s}}(it)) && \text{(2b)} \\
\wedge \quad & EqAll(end_{\mathtt{s}}, tp_s(nl_{\mathtt{s}})) && \text{(2c)}
\end{aligned}$$

4.5 Trace Lemma Reasoning

Trace logic $\mathcal{L}$ allows one to naturally express common program behavior over timepoints. Specifically, it allows us to reason about (i) all iterations of a loop, and (ii) the existence of specific timepoints. In [11], we leveraged such reasoning with the use of so-called *trace lemmas*, capturing common inductive properties of program loops. Trace lemmas are instances of the schema of bounded induction for natural numbers

$$\begin{aligned}\Big(P(bl) \wedge \forall x_{\mathbb{N}}.\big((bl \le x < br \wedge P(x)) \rightarrow P(\mathtt{suc}(x))\big)\Big) \rightarrow \\ \forall x_{\mathbb{N}}.\big(bl \le x < br \wedge P(x)\big)\end{aligned} \tag{3}$$

An example of a trace lemma would be the statement formalising that a certain program variable's value remains unchanged from a specific iteration to the end of loop execution. In this work, instead of adding instances of (3) statically to strengthen loop semantics, we move induction into the first-order prover. The advantage of adding instances of (3) dynamically is that during proof search we have more information available and can thus perform induction in a more controlled and goal oriented fashion.

Nonetheless, due to some limitations in our first-order prover, we are unable to completely do away with additional lemmas. Specifically, we need to nudge the prover to deduce that a loop counter expression will, at the end of loop execution, have the value of the expression it is compared against in the loop condition.

(A) Equal Lengths Trace Lemma We define a common property of loop counter expressions. We call a program expression `e` *dense* at loop `w` if:

$$Dense_{w,e} := \forall it_{\mathbb{N}}.\Big(it < nl_{\mathtt{w}} \rightarrow \begin{pmatrix} [\![\mathtt{e}]\!](tp_{\mathtt{w}}(\mathtt{suc}(it))) \simeq [\![\mathtt{e}]\!](tp_{\mathtt{w}}(it)) \vee \\ [\![\mathtt{e}]\!](tp_{\mathtt{w}}(\mathtt{suc}(it))) \simeq [\![\mathtt{e}]\!](tp_{\mathtt{w}}(it)) + 1 \end{pmatrix}\Big).$$

Let `w` be a while-statement, $C_{\mathtt{w}} :=$ `e < e'` be the loop condition where `e'` is a program expression that remains constant during iterations of `w`. The *equal lengths trace lemma of* `w`*,* `e` *and* `e'` is defined as

$$\begin{aligned}\big(Dense_{w,e} \wedge [\![\mathtt{e}]\!](tp_{\mathtt{w}}(\mathtt{0})) \le [\![\mathtt{e'}]\!](tp_{\mathtt{w}}(\mathtt{0}))\big) \rightarrow \\ [\![\mathtt{e}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}})) \simeq [\![\mathtt{e'}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}})).\end{aligned} \tag{A}$$

Trace lemma A states that a dense expression `e` smaller than or equal to some expression `e'` that does not change in the loop, will eventually, specifically in the last iteration, reach the same value as `e'`. This follows from the fact that we assume termination of a loop, hence we assume the existence of a timepoint $nl_{\mathtt{w}}$ where the loop condition does not hold anymore. As a consequence, given that the loop condition held at the beginning of the execution, we can derive that the loop counter value immediately after the loop execution $[\![\mathtt{e}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}}))$ will necessarily equate to $[\![\mathtt{e'}]\!](tp_{\mathtt{w}}(\mathtt{0})) = [\![\mathtt{e'}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}}))$. Note that a similar lemma can just as easily be added for dense but decreasing loop counters.

5 Multi-Clause Goal Induction for Lemmaless Induction

As mentioned above, the main focus of our work is moving induction into the saturation prover. We achieve this by adding inference rules that apply induction to loop counter terms. We leverage recent theorem proving effort on *bounded (integer) induction* in saturation [14,15]. However, as illustrated in the following, these recent efforts cannot be directly used in trace logic reasoning since we need to (i) adjust bounded induction for the setting of natural numbers, and (ii) generalise to multi-clause induction. We discuss these steps using Fig. 1. Verifying the safety assertion of Fig. 1 requires proving the trace logic formula:

$$\begin{aligned}&\forall pos_{\mathbb{I}}.\, \exists j_{\mathbb{I}}.\, (0 \leq pos < (2 \times length)\\ &\quad \rightarrow (c(main\_end, pos) \simeq a(j) \vee c(main\_end, pos) \simeq b(j))\end{aligned} \tag{4}$$

For proving (4), it suffices to prove that the following, slightly modified statement is a loop invariant of Fig. 1:

$$\begin{aligned}&\forall it_{\mathbb{N}}.\, it < nl_{\mathtt{w}} \rightarrow \forall pos_{\mathbb{I}}.\, \exists j_{\mathbb{I}}.\, (0 \leq pos < (2 \times i(tp_{\mathtt{w}}(it))))\\ &\quad \rightarrow (c(tp_{\mathtt{w}}(it), pos) \simeq a(j) \vee c(tp_{\mathtt{w}}(it), pos) \simeq b(j))\end{aligned} \tag{5}$$

where `w` refers to the loop statement in Fig. 1. As part of the program semantics in trace logic, we have formula (6) which links the value of c at the end of the loop to its value at the end of the program. Moreover, using the trace lemma A, we also derive formula (7) in trace logic:

$$\forall pos_{\mathbb{I}}. c(tp_{\mathtt{w}}(nl_{\mathtt{w}}), pos) \simeq c(main\_end, pos) \tag{6}$$

$$i(tp_{\mathtt{w}}(nl_{\mathtt{w}})) \simeq length \tag{7}$$

It is tempting to think that in the presence of these clauses (6)–(7), a saturation-based prover would rewrite the negated conjecture (4) to

$$\begin{aligned}&\neg(\forall pos_{\mathbb{I}}.\, \exists j_{\mathbb{I}}.\, (0 \leq pos < (2 \times i(tp_{\mathtt{w}}(nl_{\mathtt{w}}))))\\ &\quad \rightarrow (c(tp_{\mathtt{w}}(nl_{\mathtt{w}}), pos) \simeq a(j) \vee c(tp_{\mathtt{w}}(nl_{\mathtt{w}}), pos) \simeq b(j)))\end{aligned}$$

from which a bounded natural number induction inference (similar to the $\mathtt{IntInd}_{<}$ rule of [15]) would quickly introduce an induction hypothesis with (5) as the conclusion, by induction over $nl_{\mathtt{w}}$. However, this is not the case, as most saturation provers work by first *clausifying* their input. The negated conjecture (4) would not remain a single formula, but be split into the following clauses where sk is a Skolem symbol:

$$\begin{array}{cc} a(x) \not\simeq c(main\_end, sk) & b(x) \not\simeq c(main\_end, sk)\\ \neg(sk \leq 0) & sk \leq 2 \times length \end{array}$$

These clauses can be rewritten using (6)–(7). For example, the first clause can be rewritten to $a(x) \not\simeq c(tp_{\mathtt{w}}(nl_{\mathtt{w}}, sk))$. However, attempting to prove the negation of any of the rewritten clauses individually via induction would merely result in the addition of useless induction formulas to the search space. For example, attempting to prove $\forall it_{\mathbb{N}}.\, it < nl_{\mathtt{w}} \rightarrow (\exists x_{\mathbb{I}}.\, a(x) \simeq c(tp_{\mathtt{w}}(it), sk))$, is pointless as it is clearly false. *The solution we propose in this work is to use multi-clause induction*, whereby we attempt to prove the negation of the conjunction of multiple clauses via a single induction inference. For our running example Fig. 1, we can use the following rewritten versions of clauses from the negated conjecture $a(x) \not\simeq c(tp_{\mathtt{w}}(nl_{\mathtt{w}}, sk))$, $b(x) \not\simeq c(tp_{\mathtt{w}}(nl_{\mathtt{w}}, sk))$, and $sk \leq 2 \times i(tp_{\mathtt{w}}(nl_{\mathtt{w}}))$, with induction term $nl_{\mathtt{w}}$, to obtain the induction formula:

$$\begin{array}{l} \neg\Big(\quad \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathtt{w}}(0)), sk) \\ \quad \wedge\ \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathtt{w}}(0), sk)) \\ \quad \wedge\ sk \leq 2 \times i(tp_{\mathtt{w}}(0))\Big) \\ \wedge\ StepCase \end{array} \rightarrow \begin{array}{l} \forall it_{\mathbb{N}}.\, it < nl_{\mathtt{w}} \rightarrow \\ \quad \neg\Big(\ \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathtt{w}}(it), sk) \\ \qquad \wedge \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathtt{w}}(it), sk) \\ \qquad \wedge\ sk \leq 2 \times i(tp_{\mathtt{w}}(it))\Big) \end{array} \tag{8}$$

where *StepCase* is the formula:

$$\begin{array}{l} \forall it_{\mathbb{N}}.\, it < nl_{\mathtt{w}} \wedge \\ \neg\Big(\quad \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathtt{w}}(it)), sk) \\ \quad \wedge\ \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathtt{w}}(it)), sk) \\ \quad \wedge\ sk \leq i(tp_{\mathtt{w}}(y))\Big) \end{array} \rightarrow \begin{array}{l} \neg\Big(\quad \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathtt{w}}(\mathtt{suc}(it)), sk) \\ \quad \wedge \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathtt{w}}(\mathtt{suc}(it)), sk) \\ \quad \wedge\ sk \leq 2 \times i(tp_{\mathtt{w}}(\mathtt{suc}(it)))\Big) \end{array}$$

Using the induction formula (8), a contradiction can then easily be derived, establishing validity of (4). In what follows, we formalize the multi-clause induction principle we used above. To this end, we introduce a generic inference rule, called *multi-clause goal induction* and denoted as `MCGLoopInd`.

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

For performance reasons, we mandate that the premises $C_1 \ldots C_n$ be derived from trace logic formulas expressing safety assertions and not from formulas encoding the program semantics. The `MCGLoopInd` rule is formalised only as an induction inference over last loop iteration symbols. While restricting to $nl_{\mathtt{w}}$ terms is of purely heuristic nature, our experiments justify the necessity and usefulness of this condition (Sect. 8).

6 Array Mapping Induction for Lemmaless Induction

Multi-clause goal induction neatly captures goal-oriented application of induction. Nevertheless, there are verification challenges where `MCGLoopInd` fails to prove inductive loop properties. This is particularly the case for benchmarks

```
    func main(){
      const Int alength;
      Int[] a;
      Int i = 0;
      const Int n;

      while(i < alength){
        a[i] = a[i] + n;
        i = i + 1;
      }

      Int j = 0;
      while(j < alength){
        a[j] = a[j] - n;
        j = j + 1;
      }
    }
  assert (∀pos_𝕀.((0 ≤ pos < alength)
         → a(main_end, pos) = a(main_start, pos)))
```

Fig. 3. Adding and subtracting `n` to every element of array `a`.

containing multiple loops, such as in Fig. 3. We first discuss the limitations of `MCGLoopInd` using Fig. 3, after which we present our solution, the *array mapping induction* inference.

Let $\mathtt{w}_1$ be the first loop statement of Fig. 3 and $\mathtt{w}_2$ be the second loop. Using `MCGLoopInd`, we would attempt to prove

$$\begin{aligned}&\forall it_{\mathbb{N}}.\, it \leq nl_{\mathtt{w}_2} \rightarrow \\ &\quad \forall pos_{\mathbb{I}}.\,(0 \leq pos < j(tp_{\mathtt{w}_2}(it))) \rightarrow (a(tp_{\mathtt{w}_2}(it), pos) \simeq a(main\_start, pos)\end{aligned} \tag{9}$$

However, formula (9) is not a useful invariant for proving the assertion. Rather, for $\mathtt{w}_2$ we need a loop invariant similar to

$$\begin{aligned}\forall it_{\mathbb{N}}.\, it \leq nl_{\mathtt{w}_2} \rightarrow \forall pos_{\mathbb{I}}.\,(0 \leq pos < j(tp_{\mathtt{w}_2}(it)))\\ \rightarrow (a(tp_{\mathtt{w}_2}(it), pos) \simeq a(tp_{\mathtt{w}_2}(0), pos) - n\end{aligned} \tag{10}$$

and a similar loop invariant for loop $\mathtt{w}_1$. The loop invariant (10) is however not linked to the safety assertion of Fig. 3, and thus multi-clause goal induction is unable to infer and prove with it. To aid with the verification of benchmarks such as Fig. 3, we introduce another induction inference which we call *array mapping induction*. In this case, we trigger induction not on clauses and terms coming from the goal, but on clauses and terms appearing in the program semantics.

The *array mapping induction* inference rule, denoted as `AMLoopInd` is given below. Essentially, `AMLoopInd` involves analysing a clause set to heuristically devise a suitable loop invariant. Guessing a candidate loop invariant is a difficult problem. The `AMLoopInd` inference is triggered if clauses of the shapes of C_1 and

C_2 defined below are present in the clause set. Intuitively, C_2 can be read as saying that on each round of some loop $\mathtt{w}$, some array a at position i is set to some function F of its previous value at that position. Clause C_1 states that i increases by m in each round of the loop. Together the two clauses suggest that the loop is mapping the function F to each mth location of the array starting from the array cell located at $i(tp_{\mathtt{w}}(0))$. This is precisely what the induction formula attempts to prove. Note that for ease of notation, we present the inference for the case where the indexing variable is *increasing*. It is straightforward to generalise to the decreasing case. The `AMLoopInd` rule is[1]

$$\frac{\begin{array}{c} C_1 = i(tp_{\mathtt{w}}(\mathtt{suc}(x))) \simeq i(tp_{\mathtt{w}}(x)) + m \;\vee\; \neg(x < nl_{\mathtt{w}}) \\ C_2 = a\big(tp_{\mathtt{w}}(\mathtt{suc}(x)), i(tp_{\mathtt{w}}(x))\big) \simeq F[a\big(tp_{\mathtt{w}}(x), i(tp_{\mathtt{w}}(x))\big)] \;\vee\; \neg(x < nl_{\mathtt{w}}) \end{array}}{\mathrm{CNF}(\mathit{StepCase} \rightarrow \mathit{Conclusion})}$$

where $\mathtt{w}$ is some loop and F an arbitrary non-empty context. Let i_0 be an abbreviation for $i(tp_{\mathtt{w}}(0))$. Then:

$$\begin{array}{ll} \mathit{StepCase}: & \forall it_{\mathbb{N}}.\ \big(\forall y_{\mathbb{I}}.\ it < nl_{\mathtt{w}} \wedge \\ & \qquad y < i(tp_{\mathtt{w}}(it)) - i_0 \wedge y \geq 0 \wedge y \;\mathtt{mod}\; m = 0 \\ & \qquad\qquad \rightarrow a(tp_{\mathtt{w}}(it), i_0 + y) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + y)]\big) \rightarrow \\ & (\forall y_{\mathbb{I}}.\ y < i(tp_{\mathtt{w}}(\mathtt{suc}(it))) - i_0 \wedge y \geq 0 \wedge y \;\mathtt{mod}\; m = 0 \\ & \qquad\qquad \rightarrow a(tp_{\mathtt{w}}(\mathtt{suc}(it)), i_0 + y) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + y)]) \\ \mathit{Conclusion}: & \forall x_{\mathbb{I}}.\ x < i(tp_{\mathtt{w}}(nl_{\mathtt{w}})) - i_0 \wedge x \geq 0 \wedge x \;\mathtt{mod}\; m = 0 \\ & \qquad\qquad \rightarrow a(tp_{\mathtt{w}}(nl_{\mathtt{w}}), i_0 + x) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + x)] \end{array}$$

To prove *StepCase*, it is necessary to be able to reason that positions in the array a remain unchanged until visited by the indexing variable. This can be achieved via the addition of another induction to the conclusion of the inference. We do not provide details of this induction formula here, but it is added to the conclusion by our implementation which we present in Sect. 7. The `AMLoopInd` inference is thus sufficient to prove the assertion of Fig. 3. While `AMLoopInd` is a limited approach for guessing inductive loop invariants, we believe it can be extended towards further, more generic methods to guess invariants, as discussed in Sect. 9. We conclude this section by noting that our induction rules are sound, based on trace logic semantics. Since both rules merely add instances of the bounded induction schema for natural numbers (3) to the search space, soundness is trivial and we do not provide a proof.

7 Implementation

Our approach is implemented as an extension of the Rapid framework, using the first-order theorem prover Vampire.

[1] In the conclusion we ignore the base case of the induction formula as it is trivially true.

***Extensions to* Rapid.** Rapid takes as an input a $\mathcal{W}$ program along with a property expressed in $\mathcal{L}$. It outputs the semantics of the program expressed in $\mathcal{L}$ using SMT-LIB syntax along with the property to be proven. For our "lemmaless induction" framework, we have extended Rapid as follows. Firstly, we prevent the output of all trace lemmas other than trace lemma A (Sect. 4.5). We added custom extensions to the SMT-LIB language to identify trace logic symbols, such as loop iteration symbols, program variables, within the Rapid encodings. This way, trace logic symbols to be used for induction inferences are easily identified and can also be used for various proving heuristics. We refer to this version (available online[2]) as Rapid^{l-}.

***Extensions to* Vampire.** We implemented the `MCGLoopInd` inference rule and a slightly simplified version of the `AMLoopInd` rule in a new branch of Vampire[3]. The main issue with the induction inferences `MCGLoopInd` and `AMLoopInd` is their explosiveness which can cause proof search to diverge. We have, therefore, introduced various heuristics in the implementation to try and control them. For `MCGLoopInd` we not only necessitate that the premises are derived from the conjecture, but that their derivation length from the conjecture is below a certain distance controlled by an option. The premises must be unit clauses unless another option `multi_literal_ clauses` is toggled on. The option `induct_all_loop_counts` allows `MCGLoopInd` induction to take place on all loop counter terms, not just final loop iterators. In order for the `MCGLoopInd` and `AMLoopInd` inferences to be applicable, we need to rewrite terms not containing final loop counters to terms that do. However, rewriting in Vampire is based on superposition, which is parameterised by a term order preventing smaller terms to be rewritten into larger ones. In this case, the term order may work against us and prevent such rewrites from happening. We implemented a number of heuristics to handle this problem. One such heuristic is to give terms representing constant program variables a large weight in the ordering. Then, equations such as $alength \simeq i(tp_{\mathtt{w}}(nl_{\mathtt{w}}))$ will be oriented left to right as desired. We combined these options with others to form a portfolio of strategies[4] that contains 13 strategies each of which runs in under 10s.

8 Experimental Results

Benchmarks. For our experiments, we use a total of 111 examples whose verification involved proving safety assertions of different logical complexity (quantifier-free, only universally/existentially quantified, and with quantifier alternations). Our benchmarks are divided into four groups, as indicated in Table 1: (i) the first 13 problems have quantifier-free proof obligations; (ii) the majority of benchmarks, in total 68 examples, contain universally quantified

[2] See commit `285e54b7e` of https://github.com/vprover/rapid/tree/ahmed-induction-support.

[3] See commit `4a0f319f` of https://github.com/vprover/vampire/tree/ahmed-rapid.

[4] `--mode portfolio --schedule rapid_induction.`.

Table 1. Experimental results.

Benchmark	(1)	(2)	(3)	(4)
atleast_one_iteration_0	✓	✓	✓	✓
atleast_one_iteration_1	✓	✓	✓	✓
count_down	✓	-	-	-
eq	✓	-	✓	-
find_sentinel	✓	✓	-	-
find1_0	✓	✓	✓	-
find1_1	✓	✓	✓	-
find2_0	✓	✓	✓	-
find2_1	✓	✓	✓	-
indexn_is_arraylength_0	✓	✓	✓	-
indexn_is_arraylength_1	✓	✓	✓	-
set_to_one	✓	✓	✓	✓
str_cpy_3	✓	✓	✓	-
add_and_subtract	✓	-	-	✓
both_or_none	✓	✓	-	✓
check_equal_set_flag_1	✓	✓	-	✓
collect_indices_eq_val_0	✓	✓	-	✓
collect_indices_eq_val_1	✓	✓	-	-
copy	✓	✓	-	✓
copy_absolute_0	✓	✓	-	✓
copy_absolute_1	✓	✓	-	✓
copy_and_add	✓	-	-	✓
copy_nonzero_0	✓	✓	-	✓
copy_partial	✓	✓	-	✓
copy_positive_0	✓	✓	-	✓
copy_two_indices	✓	✓	-	-
find_max_0	✓	✓	-	✓
find_max_2	✓	✓	-	✓
find_max_from_second_0	✓	-	-	✓
find_max_local_2	-	-	-	-
find_max_up_to_0	-	-	-	-
find_max_up_to_2	-	-	-	-
find_min_0	✓	✓	-	✓
find_min_2	✓	✓	-	-
find_min_local_2	-	-	-	-
find_min_up_to_0	-	-	-	-
find_min_up_to_2	-	-	-	-
find1_4	-	✓	-	-
find2_4	✓	✓	-	-
in_place_max	✓	✓	-	✓
inc_by_one_0	✓	✓	-	✓
inc_by_one_1	✓	✓	-	✓
inc_by_one_harder_0	✓	✓	-	✓
inc_by_one_harder_1	✓	✓	-	✓
init	✓	✓	-	-
init_conditionally_0	✓	✓	-	-
init_conditionally_1	✓	✓	-	✓
init_non_constant_0	✓	✓	-	-
init_non_constant_1	✓	✓	-	✓
init_non_constant_2	✓	✓	-	✓
init_non_constant_3	✓	✓	-	✓
init_non_constant_easy_0	✓	✓	-	-
init_non_constant_easy_1	✓	✓	-	✓
init_non_constant_easy_2	✓	✓	-	✓
init_non_constant_easy_3	✓	✓	-	✓
init_partial	✓	✓	-	✓
init_prev_plus_one_0	✓	✓	-	-
init_prev_plus_one_1	✓	✓	-	-
init_prev_plus_one_alt_0	✓	✓	-	-
init_prev_plus_one_alt_1	✓	✓	-	-
insertion_sort	-	-	-	-
max_prop_0	✓	✓	-	✓
max_prop_1	✓	✓	-	✓
merge_interleave_0	✓	-	-	✓
merge_interleave_1	✓	-	-	✓
min_prop_0	✓	✓	-	✓
min_prop_1	✓	✓	-	✓
partition_0	✓	✓	-	✓
partition_1	✓	✓	-	✓
push_back	✓	✓	-	✓
reverse	✓	✓	-	-
rewnifrev	✓	-	-	✓
rewrev	✓	-	-	✓
skipped	✓	-	-	✓
str_cpy_0	✓	✓	-	-
str_cpy_1	✓	✓	-	-
str_cpy_2	✓	✓	-	-
swap_0	-	✓	✓	✓
swap_1	-	✓	✓	✓
vector_addition	✓	✓	-	✓
vector_subtraction	✓	✓	-	✓
check_equal_set_flag_0	✓	✓	-	-
find_max_1	-	-	-	-
find_max_from_second_1	✓	-	-	-
find1_2	✓	✓	-	-
find1_3	✓	✓	-	-
find2_2	✓	✓	-	-
find2_3	✓	✓	-	-
collect_indices_eq_val_2	-	✓	-	-
collect_indices_eq_val_3	✓	-	-	-
copy_nonzero_1	✓	✓	-	-
copy_positive_1	✓	✓	-	-
find_max_local_0	-	-	-	-
find_max_local_1	✓	-	-	-
find_max_up_to_1	-	-	-	-
find_min_1	-	-	-	-
find_min_local_0	-	-	-	-
find_min_local_1	✓	-	-	-
find_min_up_to_1	-	-	-	-
merge_interleave_2	✓	-	-	-
partition_2	✓	✓	-	-
partition_3	✓	✓	-	-
partition_4	-	-	-	-
partition_5	-	✓	-	-
partition_6	-	-	-	-
partition-harder_0	✓	✓	-	-
partition-harder_1	✓	✓	-	-
partition-harder_2	✓	-	-	-
partition-harder_3	✓	-	-	-
partition-harder_4	✓	-	-	-
str_len	✓	✓	-	-
Total solved	**93**	**78**	**13**	**47**

safety assertions; (iii) 7 problems come with the task of verifying existentially quantified assertions; (iv) and the last 23 programs contain assertions with alternation of quantifiers. The examples from (i)-(ii), a total of 81 programs, come from the array verification benchmarks of SV-COMP repository [1], with most of these examples originating from [9,13].[5] These examples correspond to the set of those SV-COMP benchmarks which use the C fragment supported by Rapid; specifically, when selecting examples (i)-(ii) from SV-COMP, we omitted examples containing pointers or memory management. All SV-COMP examples from (i)-(ii) are adapted to our input format, as for example arrays in trace logic are treated as unbounded data structures. Further, the examples (iii)-(iv) are new examples crafted by us, in total 30 new examples. They contain existential and alternating quantification in safety assertions. We intend to submit these 30 examples from (iii)-(iv) to SV-COMP.

Experimental Setting. We used two versions of Rapid in our experiments. First, `(1)` Rapid^{l-} denotes our Rapid approach, using lemmaless induction `MCGLoopInd` and `AMLoopInd` in Vampire. Further, `(2) Rapid`$^{l+}$ uses trace lemmas for inductive reasoning, as described in [11]. We also compared Rapid^{l-} with other verification tools. In particular, we considered `(3)` SeaHorn and `(4)` Vajra (and its extension Diffy that produced for us exactly the same results as Vajra). SeaHorn converts the program into a constrained horn clause (CHC) problem and uses the SMT solver Z3 for solving. Vajra and Diffy implement inductive reasoning and recurrence solving over loop counters; in the background, they also use Z3.

Rapid ***Experiments.*** Table 1 shows that Rapid^{l-} is superior to Rapid^{l+}, as it solves a total of 93 problems, while Rapid^{l+} only proved 78 assertions correct. Particularly, Rapid^{l-} can solve benchmark `merge_interleave_2` corresponding to our motivating example 1, and other challenging problems such as `find_max_local_1` also containing quantifier alternations.

While Rapid^{l-} can solve a total of ten problems more than Rapid^{l+}, it is interesting to look into which problems can now be solved. Many of the newly solved problems are structurally very close to the loop invariants needed to prove them. This is where multi-clause goal-oriented induction `MCGoalInd` makes the biggest impact. For instance, this allows Rapid^{l-} to prove the partial correctness of `find_max_ from_second_0` and `find_max_from_second_1`.

On the other hand, Rapid^{l-} also lost two challenging benchmarks that were previously solved by Rapid^{l+}, namely `swap_0` and `partition_5`. This could be for two reasons: (1) the strategies in the induction schedule of Rapid^{l-} are too restrictive for such benchmarks, or (2) the step case of the induction axiom introduced by our two rules are too difficult for Vampire to prove. Strengthening lemmaless induction with additional trace lemmas from Rapid^{l+} is an interesting line of further work.

[5] Artifact evaluation: in order to reproduce the results reported in this section, please follow the instructions at https://github.com/vprover/vampire_publications/tree/master/experimental_data/CICM-2022-RAPID-INDUCTION.

Comparing with other tools. Both, SeaHorn and Vajra/Diffy require C code as input, whereas Rapid uses its own syntax. We translated our benchmarks to C code expressing the same problem. However, a direct comparison of Rapid, and in particular Rapid^{l-}, with most other verifiers requiring standard C code as an input is not possible as we consider slightly different semantics. In contrast to SeaHorn and Vajra/Diffy, we assume that integers and arrays are unbounded and that all array positions are initialized by arbitrary data. Further, we can read/write at any array position without allocating the accessed memory beforehand. Apart from semantic differences, Rapid can directly express assertions and assumptions containing quantifiers and put variable contents from different points in time into relation. In order to deal with the latter, we introduced history variables in the code provided to SeaHorn and Vajra/Diffy. Quantification was simulated by non-deterministically assigned variables and by loops. As a result, SeaHorn verified 13 examples, whereas Vajra/Diffy 47 of our benchmarks. As Vajra/Diffy restrict their input programs to contain only loops having very specific loop-conditions, several of our benchmarks failed. For example, $i < length$ is permitted, whereas $a[i] \neq 0$ is not. Vajra/Diffy could prove correctness for nearly all the programs satisfying these restrictions. SeaHorn, on the other hand, has problems with the complexity introduced by the arrays. It could solve especially those benchmarks whose correctness do not depend on the arrays' content.

9 Future Directions and Conclusion

We introduced lemmaless induction to fully automate the verification of inductive properties of program loops with unbounded arrays and integers. We introduced goal-oriented and array mapping induction inferences, triggered by loop counters, in superposition-based theorem proving. Our results show that lemmaless induction in trace logic outperforms other state-of-the-art approaches in the area. There are various ways to further develop lemmaless induction in trace logic. On larger benchmarks, particularly those containing multiple loops, our approach struggles. For loops where the required invariant is not connected to the conjecture, we introduced array mapping induction. However, the array mapping induction inference is limited in the form of invariants it can generate. We would like to investigate other methods, such as machine learning for synthesising loop invariants that are not too prolific. A completely different line of research that we are currently working on, is updating the trace logic syntax and semantics of $\mathcal{W}$ to deal with memory and memory allocation, aiming to efficiently reason about loop operations over the memory.

As shown in [18], the validity problem for first-order formulas of linear arithmetic extended with non-theory function symbols is Π_1^1-complete. Therefore, we do not expect any completeness result for inductive theorem proving. Proving relative completeness results for our verification framework is an interesting question.

Acknowledgements. This research was partially supported by the ERC consolidator grant ARTIST 101002685, the FWF research project LogiCS W1255-N23, the TU Wien SecInt doctoral program, and the EUProofNet Cost Action CA20111. Our research was partially funded by the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

References

1. SV-comp repository. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks
2. Vampire website. https://vprover.github.io/
3. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (Eds.) Handbook of Automated Reasoning, vol. I, chap. 2, pp. 19–99. Elsevier Science (2001)
4. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction. In: TACAS 2020. LNCS, vol. 12078, pp. 22–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_2
6. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: inductive reasoning of array programs using difference invariants. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 911–935. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_42
7. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 392–406. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_27
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_14
10. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 259–277. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_14
11. Georgiou, P., Gleiss, B., Kovács, L.: Trace logic for inductive loop reasoning. In: 2020 Formal Methods in Computer Aided Design (FMCAD), pp. 255–263. IEEE (2020)
12. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
13. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 248–266. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_15
14. Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: Benzmüller, C., Miller, B. (eds.)

CICM 2020. LNCS (LNAI), vol. 12236, pp. 123–137. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53518-6_8
15. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 361–377. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_21
16. Karpenkov, E.G., Monniaux, D.: Formula slicing: inductive invariants from preconditions. In: Bloem, R., Arbel, E. (eds.) HVC 2016. LNCS, vol. 10028, pp. 169–185. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49052-6_11
17. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common Lisp. In: IEEE Transactions on Software Engineering, pp. 203–213 (1997)
18. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74915-8_19
19. Kovács, L., Robillard, S., Voronkov, A.: Coming to terms with quantified reasoning. In: POPL, pp. 260–270 (2017)
20. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
21. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: SMT-based array invariant generation. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 169–188. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_12
22. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
23. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 38–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_3
24. Bjoner, N., Reger, G., Suda, M., Voronkov, A.: AVATAR modulo theories. In: GCAI, pp. 39–52 (2016)
25. Reger, G., Schoisswohl, J., Voronkov, A.: Making theory reasoning simpler. In: TACAS 2021. LNCS, vol. 12652, pp. 164–180. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_9
26. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, pp. 223–234 (2009)

Formal Methods in Computer-Aided Design 2022

The RAPID Software Verification Framework

Pamina Georgiou[a], Bernhard Gleiss[a], Ahmed Bhayat[b], Michael Rawson[a], Laura Kovács[a], Giles Reger[b]
[a] TU Wien, Vienna, Austria
[b] University of Manchester, Manchester, United Kingdom
{ pamina.georgiou, bernhard.gleiss, michael.rawson, laura.kovacs }@tuwien.ac.at, { ahmed.bhayat, giles.reger }@manchester.ac.uk

***Abstract*—We present the RAPID framework for automatic software verification by applying first-order reasoning in *trace logic*. RAPID establishes partial correctness of programs with loops and arrays by inferring invariants necessary to prove program correctness using a saturation-based automated theorem prover. RAPID can heuristically generate *trace lemmas*, common program properties that guide inductive invariant reasoning. Alternatively, RAPID can exploit nascent support for induction in modern provers to fully automate inductive reasoning without the use of trace lemmas. In addition, RAPID can be used as an invariant generation engine, supplying other verification tools with quantified loop invariants necessary for proving partial program correctness.**

I. INTRODUCTION

State-of-the-art deductive verification tools for programs containing inductive data structures ([1], [2], [3], [4], [5]) largely depend on satisfiability modulo theories (SMT) solvers to discharge verification conditions and establish software correctness. These approaches are mostly limited to reasoning over universally-quantified properties in fragments of first-order *theories*: arrays, integers, etc. In contrast, RAPID supports reasoning with arbitrary quantifiers in full first-order logic with theories [6]. Program semantics and properties are directly encoded in trace logic by quantifying over *timepoints* of program execution. This allows simultaneous reasoning about *sets* of program states, unlike model-checking approaches [2][7]. The gain in expressiveness is beneficial for reasoning about programs with unbounded arrays [6] or to prove security properties [8], for example.

This paper presents what RAPID can do, sketches its design (Section III), and describes its main components and implementation aspects (Sections IV–VII). Experimental evaluation using the SV-COMP benchmark [9] shows RAPID's efficacy in verification (Section VIII).

Given a program loop annotated with pre/post-conditions, RAPID offers two modes for proving partial program correctness. In the first, RAPID relies on so-called *trace lemmas*, apriori-identified inductive properties that are automatically instantiated for a given program. In the second, RAPID delegates inductive reasoning to the underlying first-order theorem prover [10][11], without instantiating trace lemmas. In either mode, the automated theorem prover used by RAPID is VAMPIRE [12]. RAPID can also synthesize quantified invariants from program semantics, complementing other invariant-generation methods.

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

Fig. 1: Program partitioning an array `a` into two arrays `b, c` containing positive and negative elements of `a` respectively.

Related Work: Verifying programs with unbounded data structures can use model checking for invariant synthesis. Tools like Spacer/Quic3 ([4], [2]), SEAHORN [1] or FREQHORN [7] are based on constrained horn clauses (CHC) and use either fixed-point calculation or sampling/enumerating invariants until a given safety assertion is proved. These approaches use SMT solvers to check validity of invariants and are limited to quantifier-free or universally-quantified invariants. Recurrence solving and data-structure-specific tactics can be used to infer and prove quantified program properties [3]. DIFFY [13] and VAJRA [5] derive relational invariants of two mutations of a program such that inductive properties can be enforced over the entire program, without invariants for each individual loop.

II. MOTIVATING EXAMPLE

We motivate RAPID using the program in Figure 1, written in a standard while-like programming language $\mathcal{W}$. Each program in $\mathcal{W}$ consists of a single top-level function `main`, with arbitrary nestings of if-then-else and while statements. $\mathcal{W}$ includes optionally-mutable integer (array) variables, and standard side-effect-free expressions over Booleans and integers.

Semantics and properties of $\mathcal{W}$-programs are expressed in *trace logic* $\mathcal{L}$, an instance of many-sorted first-order logic with theories and equality [6]. A *timepoint* in trace logic is a term of sort $\mathbb{L}$ that refers to a program location. For example, l_5 refers to `line 5` in Figure 1. If a program location occurs in a loop, a timepoint is represented by a function $l : \mathbb{N} \mapsto \mathbb{L}$, where the argument is a natural number representing a loop iteration.

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_32

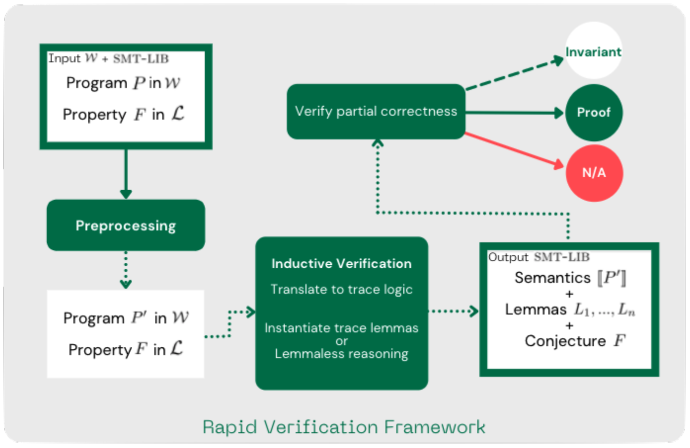
Input $\mathcal{W}$ + SMT-LIB
Program P in $\mathcal{W}$
Property F in $\mathcal{L}$
Preprocessing
Program P' in $\mathcal{W}$
Property F in $\mathcal{L}$
Inductive Verification
Translate to trace logic
Instantiate trace lemmas
or
Lemmaless reasoning
Output SMT-LIB
Semantics $[\![P']\!]$
+
Lemmas $L_1, ..., L_n$
+
Conjecture F
Verify partial correctness
Invariant
Proof
N/A
Rapid Verification Framework

Fig. 2: Overview of the RAPID verification framework.

For example, $l_6(0)$ denotes the first iteration of the loop before entering the loop body. A mutable scalar variable v is modeled as a function over time $v : \mathbb{L} \mapsto \mathbb{I}$. An array variable is modeled as a function $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$, where array indices are represented by integer arguments. For constant variables we omit the timepoint argument. We use a constant $nl_i : \mathbb{N}$ to denote the last iteration of the loop starting at l_i. When a loop is nested within other loops, the last iteration is a *function* over timepoints of all enclosing loops; l_{end} denotes the timepoint after program execution. For Figure 1, $l_6(nl_6)$ denotes the program location of the loop at its last iteration, when the loop condition no longer holds. We assume that programs terminate, and hence RAPID focuses on *partial* correctness.

Figure 1 creates two new arrays, `b` and `c`, containing positive and negative elements from the input array `a` respectively. Note that the arrays are unbounded, and we use the symbolic, non-negative constant `alength` to bound the length of the input array `a`. The constraint that `alength` be non-negative can be expressed within a conjecture (see (1) below for example). A safety property we want to check is that for any position in `b` there exists a position in `a` such that both values are equal within the respective array bounds (and similarly for `c`). This equates to the following conjecture expressed in trace logic[1]:

$$\forall pos_{\mathbb{I}}.\ \exists pos'_{\mathbb{I}}.\ 0 \leq pos < blength(l_{end}) \land alength \geq 0 \rightarrow 0 \leq pos' < alength \land b(l_{end}, pos) = a(pos'), \tag{1}$$

To the best of our knowledge other verification approaches cannot automatically validate (1) due to quantifier alternation, but RAPID proves this property for Figure 1.

III. THE RAPID FRAMEWORK

The RAPID framework consists of approximately 10,000 lines of C++ [2]. Figure 2 summarizes the RAPID workflow. Inputs to RAPID are programs P written in $\mathcal{W}$ along with properties F expressed in $\mathcal{L}$. *Preprocessing* in RAPID applies program transformations for common loop-altering programming constructs, as well as timepoint inlining to obtain a simplified program P' from P (see Section IV).

Next, RAPID performs *inductive verification* (see Section V) by generating the axiomatic semantics $[\![P']\!]$ expressed in $\mathcal{L}$ and instantiating a set $L_1, ..., L_n$ of inductive properties — so-called *trace lemmas* — for the respective program variables of P'. For establishing some property F, RAPID supports two modes of inductive verification: *standard* and *lemmaless* mode. The difference in both versions relates to the underlying support for automating inductive reasoning while proving F. The *standard* verification mode equips the verification task with the trace lemmas $L_1, ..., L_n$, providing the necessary induction schemes for proving F. The *lemmaless* verification mode uses built-in inductive reasoning and relies less, or not at all, on trace lemmas. In either mode, the verification tasks of RAPID are encoded in the SMT-LIB format. Finally, a third and recent RAPID mode can be used for invariant generation (see Section VII). In this mode, RAPID "only" outputs quantified invariants using the SMT-LIB syntax; these invariants can further be used by other verification tools.

[1]we write $\forall x_S.\ F$ or $\exists x_S.\ F$ to mean that x has sort S in F
[2]available at https://github.com/vprover/rapid

```
    while(i < alength) {
      if (a[i] == x) {
        break;
      }
      i = i + 1;
    }

```

```
    Bool break = false;
    while(i < alength && !break ) {
      if (a[i] == x) {
        break = true;
      }
      if (!break) {
        i = i + 1;
      }
    }

```

Fig. 3: Loop tranformation for **break**-statement.

IV. PREPROCESSING IN RAPID

a) Program Transformations: We use standard program transformations to translate away **break**, **continue** and **return** statements. For these, RAPID introduces fresh Boolean program variables indicating whether a statement has been executed. The program is adjusted accordingly: **return** statements end program execution; **break** statements invalidate the first enclosing loop condition; and for **continue** the remaining code of the first enclosing loop body is not executed.

***Example** 1:* Figure 3 shows a standard transformation for a **break**-statement.

b) Timepoint Inlining: RAPID uses SSA-style inlining [14], [15], [16] for timepoints to simplify axiomatic program semantics and trace lemmas of a verification task. Specifically, RAPID caches (i) for each integer variable the current program

```
a = a + 2;
b = 3;
c = a + b;

assert (a(l_end) < c(l_end))
```

(a) block assignments

```
if (x < 1) {
  x = 0;
} else {
  skip;
}
while (y > 0) {
  y = y - 1;
}

assert (x(l_end) ≥ 0)
```

(b) simple branching

Fig. 4

expression assigned to it, and (ii) for each integer-array variable the last timepoint where it was assigned. Cached values are used during traversal of the program tree to simplify later program expressions. Thus we avoid defining irrelevant equalities of program variable values over unused timepoints, and only reference timepoints relevant to the property. We illustrate this on two examples:

***Example** 2 (Inlining assigned integer expressions):* The effect of inlined semantics can be observed when we encounter block assignments to integer variables: we can skip assignments and use the last assigned expression directly in any reference to the original program variable. Consider the partial program in Figure 4a. Our axiomatic semantics in trace logic [6] would result in

$$\begin{aligned}
a(l_2) = a(l_1) + 2 \quad &\wedge \quad b(l_2) = b(l_1) & \wedge\\
c(l_2) = c(l_1) \quad &\wedge \quad a(l_3) = a(l_2) & \wedge\\
b(l_3) = 3 \quad &\wedge \quad c(l_3) = c(l_2) & \wedge\\
a(l_{end}) = a(l_3) \quad &\wedge \quad b(l_{end}) = b(l_3) & \wedge\\
c(l_{end}) = a(l_3) + b(l_3) &
\end{aligned}$$

whereas the inlined version of semantics is drastically shorter:

$$a(l_{end}) = a(l_1) + 2 \qquad \wedge \qquad c(l_{end}) = (a(l_1) + 2) + 3.$$

In contrast to the extended semantics that define all program variables for each timepoint, the inlined version only considers the values of referenced program variables at the timepoint of their last assignment. Thus, when c is defined, RAPID directly references the (symbolic) values assigned to a and b. While b is not defined at all, note that a *is* defined as $a(l_{end})$ is referenced in the conjecture. Furthermore, the inlined semantics only make use of two timepoints, l_1, and l_{end}, as the remaining timepoints are irrelevant to the conjecture.

***Example** 3 (Inlining equalities with branching.):* Figure 4b shows another program that benefits from inlining equalities, as well as only considering timepoints relevant to the conjecture. The original semantics defines program variables x and y for all program locations: l_1, l_2, l_3, l_4, $l_6(it)$, $l_6(nl_6)$, l_{end}, for some iteration it and final iteration nl_6. While the program contains two variables x and y, only x is used in the property we want to prove. Since no assignments to x contain references to y, the loop semantics do not interfere with x, so we have

$$\begin{aligned}
&x(l_3) < 1 \to x(l_6(0)) = 0 & \wedge\\
&x(l_3) \geq 1 \to x(l_6(0)) = x(l_3) & \wedge\\
&\quad x(l_{end}) = x(l_6(0)) &
\end{aligned}$$

where the semantics of the loop defining y are omitted. Note that all timepoints of the if-then-else statements are flattened into the timepoint at the beginning of the loop at l_6 in iteration 0. The axiomatic semantics thus reduce to three conjuncts defining the value of x throughout the execution. However, x is not defined in any loop iteration other than the first as they are irrelevant to the property.

c) User-defined input: RAPID is fully automated. However, it may still benefit from manually-defined invariants to support the prover. Users can therefore extend the input to RAPID with first-order axioms written in the SMT-LIB format.

V. Inductive Verification in Rapid

As mentioned above, RAPID implements two verification modes; in the default *standard* mode, RAPID uses trace lemmas to prove inductive properties of programs. In its *lemmaless* mode RAPID relies on built-in induction support in saturation-based first-order theorem proving. In this section we elaborate on both modes further.

A. Standard Verification Mode: Reasoning with Trace Lemmas

RAPID's *standard* mode relies on trace lemma reasoning to automate inductive reasoning. Trace lemmas are sound formulas that are: (i) derived from bounded induction over loop iterations; (ii) represent common inductive program properties for a set of similar input programs; and (iii) are automatically instantiated for all relevant program variables of a specific input program during its translation to trace logic; see [6].

In all of our experiments from Section VIII, including the example from Figure 1, we only instantiate three generic inductive trace lemmas to establish partial correctness. One such trace lemma asserts, for example, that a program variable is not mutated after a certain execution timepoint.

***Example** 4:* Consider the safety assertion (1) of our running example from Figure 1. In its standard verification mode, RAPID proves correctness of (1) by using, among others, the following trace lemma instance

$$\begin{aligned}
&\forall j_{\mathbb{I}}.\ \forall b_{L\mathbb{N}}.\ \forall b_{R\mathbb{N}}.\Big(\\
&\qquad \forall it_{\mathbb{N}}.\Big((b_L \leq it < b_R \ \wedge\ b(l_9(b_L), j) = b(l_9(it), j))\\
&\qquad \to b(l_9(b_L), j) = b(l_9(s(it)), j)\Big)\\
&\quad \to \big(b_L \leq b_R \to b(l_9(b_L), j) = b(l_9(b_R), j)\big)\Big),
\end{aligned}$$

stating that the value of b at some position j is unchanged between two bounds b_L and b_R if, for any iteration it and its successor $s(it)$, values of b are unchanged.

Multitrace Generalization: RAPID can also be used to prove k-safety properties over k traces, useful for security-related hyperproperties such as non-interference and sensitivity [8]. For such problems it is sufficient to extend program variables to functions over time and trace, such that program variables are represented as $(\mathbb{L} \times \mathbb{T} \mapsto \mathbb{I})$. Program locations, and hence timepoints, are similarly parameterized by an argument of sort $\mathbb{T}$ to denote the same timepoint in different executions.

B. Lemmaless Verification Mode

When in *lemmaless* mode RAPID does not add any trace lemma to its verification task but relies on first-order theorem proving to derive inductive loop properties. An extended version of SMT-LIB (see Section VI) is used to provide the underlying prover with additional information to guide the search for necessary inductive schemes, such as likely symbols for induction. We further equip saturation-based theorem proving with two new inference rules that enable induction on such terms; see [17] for details. *Multi-clause goal induction* takes a formula derived from a safety assertion that contains a final loop counter, that is a symbol denoting last loop iterations, and inserts an instance of the induction schema for natural numbers with the negation of this formula as its conclusion into the proof search space. For example, consider the formula $x(l_5(nl_5)) < 0$. Multi-clause goal induction introduces the induction hypothesis $x(l_5(0)) \geq 0 \quad \wedge \quad \forall it_{\mathbb{N}}.\ (it < nl_5 \ \wedge\ x(l_5(it)) \geq 0) \rightarrow x(l_5(s(it))) \geq 0 \quad \rightarrow \quad x(l_5(nl_5)) \geq 0$. If the base and step cases can be discharged, a contradiction can be easily produced from the conclusion and original clause.

Array mapping induction also introduces an instance of the induction schema to the search space, but is not based on formulas derived from the goal. Instead, this rule uses clauses derived from program semantics to generate a suitable conclusion for the induction hypothesis.

VI. VERIFYING PARTIAL CORRECTNESS IN RAPID

For proving the verification tasks of Section V, and thus verifying partial program correctness, RAPID relies on saturation-based first-order theorem proving. To this end, each verification mode of RAPID uses the VAMPIRE prover, for which we implemented the following, RAPID-specific adjustments.

a) Extending SMT-LIB*:* Each verification task of RAPID is expressed in extensions of SMT-LIB, allowing us to treat some terms and definitions in a special way during proof search:

(i) `declare-nat`: The VAMPIRE prover has been extended with an axiomatization of the natural numbers as a term algebra, especially for RAPID-style verification purposes. We use the command `(declare-nat Nat zero s p Sub)` to declare the sort `Nat`, with constructors `zero` and successor `s`, predecessor `p` and ordering relation `Sub`.

(ii) `declare-lemma-predicate`: Our trace lemmas are usually of the form $(P_1 \wedge ... \wedge P_n) \rightarrow Conclusion_L$ for some trace lemma L with premises $P_1 \wedge ... \wedge P_n$. In terms of reasoning, it makes sense for the prover to derive the premises of such a lemma before using its conclusion to derive more facts, as we have many automatically instantiated lemmas of which we can only prove the premises of some from the semantics. To enforce this, we adapt literal selection such that inferences from premises are preferred over inferences from conclusions. Lemmas are split into two clauses $(P_1 \wedge ... \wedge P_n) \rightarrow Premise_L$ and $Premise_L \rightarrow Conclusion_L$, where $Premise_L$ is declared as a *lemma literal*. We ensure our literal selection function selects either a negative lemma literal[3] if available, or a positive lemma literal only in combination with another literal, requiring the prover to resolve premises before using the conclusion.

The *lemmaless* mode of RAPID introduces the following additional declarations to SMT-LIB:

(i) `declare-const-var`: assign symbols representing constant program variables a large weight in the prover's term ordering, allowing constant variables to be rewritten to non-constant expressions.

(ii) `declare-timepoint`: distinguish a symbol representing a timepoint from program variables, guiding VAMPIRE to apply induction upon timepoints.

(iii) `declare-final-loop-count`: declare a symbol as a final loop count symbol, eligible for induction.

b) Portfolio Modes: We further developed a collection of RAPID-specific proof options in VAMPIRE, using for example extensions of theory split queues [18] and equality-based rewritings [19]. Such options have been distilled into a RAPID portfolio schedule that can be run with `--mode portfolio -sched rapid`. Moreover, the multi-clause goal induction rule and the array mapping induction inference of RAPID have been compiled to a separate portfolio mode, accessed via `--mode portfolio -sched induction_rapid`.

VII. INVARIANT GENERATION WITH RAPID

RAPID can also be used as an invariant generation engine, synthesizing first-order invariants using the VAMPIRE theorem prover. To do so, we use a special mode of VAMPIRE to derive logical consequences of the semantics produced by RAPID. Some of these consequences may be loop invariants. The *symbol elimination* approach of [20] defined some set of program symbols undesirable, and only reports consequences that have *eliminated* such symbols from their predecessors. In RAPID, we adjust symbol elimination for deriving invariants in trace logic using VAMPIRE. These invariants may contain quantifier alternations, and some conjunction of them may well be enough to help other verification tools show some property. When RAPID is in *invariant generation* mode, the encoding of the problem is optimized for invariant generation. We limit trace lemmas to more specific versions of the bounded induction scheme. We also remove RAPID-specific symbols such as lemma literals so that they do not appear in consequences.

[3]Note that lemma literals become negative in the premise definition after CNF-transformation.

Symbol Elimination: Loop invariants should only contain symbols from the input loop language, with no timepoints. To remove such constructs, we apply symbol elimination: any symbol representing a variable `v` used on the left-hand side of an assignment is eliminated. However, we still want to generate invariants containing otherwise-eliminated variables at specific locations, so for each eliminated variable `v` we define `v_init` $= v(l_1)$ and `v_final` $= v(l_2)$ for appropriate locations l_1, l_2: these new symbols need not be eliminated.

We further adjusted symbol elimination in RAPID to output fully-simplified consequences during proof search in VAMPIRE (the so-called *active set* [12]) at the end of a user-specified time limit. Consequences that contain undesirable symbols or are pure consequences of theories are removed at this stage.

Reasoning with Integers vs. Naturals: In the standard setting, RAPID uses natural numbers (internally `Nat`) to describe loop iterations. However, in some situations it is advantageous to use the theory of integers: loop counter variable `i` of sort $\mathbb{I}$ will have the same numerical value as nl of sort $\mathbb{N}$ at the end of a loop. Integer-based timepoints allow deriving $i(l(nl)) = nl$. Such a clause can be very helpful for invariant generation, as shown in Example 5.

***Example** 5:* Consider the property $\forall x_{\mathbb{I}}.0 \leq x \leq alength \rightarrow a(x) = b(x)$. The property essentially requires us to prove that two arrays `a, b` are equal in all positions between 0 and `alength`. Such a property might for example be useful to prove when we copy from an array `b` into array `a` in a loop with loop condition $i < alength$ where `i` is the loop counter variable incremented by one in each iteration. Now when we run RAPID in the invariant generation mode, we might be able to derive a property $\forall x.0 \leq x \leq nl \rightarrow a(x) = b(x)$, essentially stating that the property holds for all iterations of the loop. The prover can further easily deduce that $i(l(nl)) \geq alength$ thanks to our semantics.

However, in case of natural numbers we cannot deduce that $i(l(nl)) = nl$ since the sorts of `i` and `nl` differ. In order to derive an invariant strong enough to prove the postcondition we depend upon the prover to find the invariant $\forall x.0 \leq x \leq i(l(nl)) \rightarrow a(x) = b(x)$ directly which cannot be deduced by the prover as our loop semantics are bounded by loop iterations rather than the loop counter values.

When using `-integerIterations on` we can circumvent this problem as the prover can then simply deduce the equality $i(l(nl)) = nl$ which makes the conjunction of clauses strong enough to prove the desired postcondition.

VIII. EXPERIMENTAL EVALUATION

We evaluated the two verification modes of RAPID and compare against the state-of-the-art solvers DIFFY and SEAHORN, as summarized below.

Benchmark Selection: Our benchmarks[4] are based on the `c/ReachSafety-Array` category of the SV-COMP repository [21], specifically from the `array-examples/*` subcategory[5] as it contains problems suitable for our input language.

[4]https://github.com/vprover/rapid/tree/main/examples/arrays

[5]https://github.com/sosy-lab/sv-benchmarks/tree/master/c/array-examples

TABLE I: Experimental Results

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

Other examples are not yet expressible in $\mathcal{W}$ due to the presence of function calls and/or unsupported memory access constructs. We manually translate all programs to $\mathcal{W}$ and express pre/post-conditions as trace logic properties. Additionally, we extend some SV-COMP examples with new conjectures containing existential and alternating quantification.

In general SV-COMP benchmarks are bounded to a certain array size N. By contrast, we treat arrays as unbounded in RAPID and reason using symbolic array lengths. Some benchmarks in the original SV-COMP repository are minor variations of each other that differ only in one concrete integer value, e.g to increment a program variable by some integer. Instead of copying each such variation for different digits, we abstract such constant values to a single symbolic integer constant such that just one of our benchmark covers numerous cases in the original SV-COMP setup.

Results: We compare our two RAPID verification modes, indicated by RAPID$_{\text{std}}$ and RAPID$_{\text{lemmaless}}$ respectively, against SEAHORN and DIFFY. All experiments were run on a cluster with two 2.5GHz 32-core CPUs with a 60-seconds timeout. Note that DIFFY produced the same results as its precursor VAJRA in this experiment. Table I summarizes our results, parentheticals indicating uniquely solved problems. Of a total of 140 benchmarks, RAPID$_{\text{std}}$ solves 91 problems, while RAPID$_{\text{lemmaless}}$ surpasses this by 12 problems. Particularly, RAPID$_{\text{lemmaless}}$ could solve more variations with quantifier alternations of our running example 1, as property-driven induction works well for such problems. A small number of instances, however, was solved by RAPID$_{\text{std}}$ but not by RAPID$_{\text{lemmaless}}$ within the time limit, indicating that trace lemma reasoning can help to fast-forward proof search. In total, RAPID solves 112 benchmarks, whereas SEAHORN and DIFFY could respectively prove 17 and 61 problems (with mostly universally quantified properties). For more detailed experimental data on subsets of these benchmarks we refer to [6], [17].

IX. CONCLUSION

We described the RAPID verification framework for proving partial correctness of programs containing loops and arrays, and its applications towards efficient inductive reasoning and invariant generation. Extending RAPID with function calls, and automation thereof, is an interesting task for future work.

Acknowledgements. This research was partially supported by the ERC consolidator grant ARTIST 101002685, the FWF research project LogiCS W1255-N23, the TU Wien SecInt doctoral program, and the EUProofNet Cost Action CA20111. Our research was partially funded by the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

References

[1] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, “The SeaHorn verification framework,” in *CAV*, 2015, pp. 343–361.

[2] A. Gurfinkel, S. Shoham, and Y. Vizel, “Quantifiers on demand,” in *ATVA*, 2018, pp. 248–266.

[3] P. Rajkhowa and F. Lin, “Extending viap to handle array programs,” in *VSTTE*, 2018, pp. 38–49.

[4] H. G. V. Krishnan, Y. Chen, S. Shoham, and A. Gurfinkel, “Global guidance for local generalization in model checking,” in *CAV*. Springer, 2020, pp. 101–125.

[5] S. Chakraborty, A. Gupta, and D. Unadkat, “Verifying array manipulating programs with full-program induction,” in *TACAS*, 2020, pp. 22–39.

[6] P. Georgiou, B. Gleiss, and L. Kovács, “Trace logic for inductive loop reasoning,” in *FMCAD*. IEEE, 2020, pp. 255–263.

[7] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, “Quantified invariants via syntax-guided synthesis,” in *CAV*, 2019, pp. 259–277.

[8] G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei, “Verifying relational properties using trace logic,” in *FMCAD*, 2019, pp. 170–178.

[9] D. Beyer, “Software verification: 10th comparative evaluation (SV-COMP 2021),” in *TACAS*, 2021, pp. 401–422.

[10] P. Hozzová, L. Kovács, and A. Voronkov, “Integer induction in saturation,” in *CADE*, 2021, pp. 361–377.

[11] G. Reger and A. Voronkov, “Induction in saturation-based proof search,” in *CADE*, 2019, pp. 477–494.

[12] L. Kovács and A. Voronkov, “First-Order Theorem Proving and Vampire,” in *CAV*, 2013, pp. 1–35.

[13] S. Chakraborty, A. Gupta, and D. Unadkat, “Diffy: Inductive reasoning of array programs using difference invariants,” in *CAV*, 2021.

[14] P. Briggs and K. D. Cooper, “Effective partial redundancy elimination,” *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 159–170, 1994.

[15] A. W. Appel, “SSA is functional programming,” *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, 1998.

[16] ——, *Modern compiler implementation in C*. Cambridge university press, 2004.

[17] A. Bhayat, P. Georgiou, C. Eisenhofer, L. Kovács, and G. Reger, “Lemmaless induction in trace logic,” Preprint, https://github.com/vprover/vampire_publications/blob/master/paper_drafts/rapid_induction.pdf.

[18] B. Gleiss and M. Suda, “Layered clause selection for theory reasoning,” in *IJCAR*, 2020, pp. 297–315.

[19] B. Gleiss, L. Kovács, and J. Rath, “Subsumption demodulation in first-order theorem proving,” in *IJCAR*, 2020, pp. 297–315.

[20] L. Kovács and A. Voronkov, “Finding loop invariants for programs over arrays using a theorem prover,” in *FASE*, 2009, pp. 470–485.

[21] SV-COMP. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks.

Sorting without Sorts

Pamina Georgiou, Márton Hajdu, and Laura Kovács

TU Wien, Austria

Abstract. We present a reasoning framework in support of software quality ensurance, allowing us to automatically verify the functional correctness of programs with recursive data structures. Specifically, we focus on functional programs implementing sorting algorithms. We formalize the semantics of recursive programs in many-sorted first-order logic while introducing sortedness/permutation properties as part of our first-order formalization. Rather than focusing on sorting lists of elements of specific first-order theories such as integer arithmetic, our list formalization relies on a sort parameter abstracting (arithmetic) theories and, hence, concrete sorts. Software validation is powered by automated theorem proving: we adjust recent efforts for automating inductive reasoning in saturation-based first-order theorem proving. Importantly, we advocate a compositional reasoning approach for fully automating the verification of functional programs implementing and preserving sorting and permutation properties over parameterized list structures. We showcase the applicability of our framework over recursive sorting algorithms, including Mergesort and Quicksort; to this end, we turn first-order theorem proving into an automated verification engine by guiding automated inductive reasoning with manual proof splits.

1 Introduction

Sorting algorithms are integrated parts of any modern programming language and, thus, ubiquitous in computing. Ensuring software quality thus naturally triggers the demand of validating the functional correctness of sorting routines. Such routines typically implement recursive/iterative operations over unbounded data structures, for instance lists or arrays, combined with arithmetic manipulations of numeric data types, such as naturals, integers or reals. Automating the formal verification of sorting routines therefore brings the challenge of automating recursive/inductive reasoning in extensions and combinations of first-order theories, while also addressing the reasoning burden arising from design choices made for the purpose of efficient sorting. Most notably, `Quicksort` [8] is known to be easily implemented when making use of recursive function calls, for example, as given in Figure 1, whereas procedural implementations of `Quicksort` require additional recursive data structures such as stacks. While `Quicksort` and other sorting routines have been proven correct by means of manual efforts [5], proof assistants [17], abstract interpreters [6], or model checkers [9], to the best of our knowledge such correctness proofs so far have not been fully automated.

```
datatype  a' list = nil | cons(a', (a' list))

quicksort :: a' list → a' list
quicksort(nil) = nil
quicksort(cons(x, xs)) =
  append(
    quicksort(filter<(x, xs)) ,
    cons(x, quicksort(filter≥(x, xs))))

append :: a' list → a' list → a' list
append(nil, xs) = xs
append(cons(x, xs), ys) = cons(x, append(xs, ys))
```

Fig. 1: Recursive functional algorithm of Quicksort, using the recursive function definitions append, $\mathtt{filter}_<$ and $\mathtt{filter}_\geq$ over lists of sort a. The datatype *list* is inductively defined by the list constructors nil and cons. Here, xs, ys denote lists whose elements are of sort a, whereas x is a list element of sort a. The append function concatenates two lists. The $\mathtt{filter}_<$ and $\mathtt{filter}_\geq$ functions return lists of elements y of xs such that $y < x$ and $y \geq x$, respectively.

In this paper we aim to verify the partial correctness of functional programs with recursive data structures, in a fully automated manner by using first-order theorem proving. The crux of our approach is a compositional reasoning setting based on superposition-based first-order theorem proving [12] with native support for induction [7] and first-order theories of recursively defined data types [11]. We extend this setting to support the first-order theory of list data structures parameterized by an abstract background theory/sort a and induction on recursive function calls - *computation induction*. We thus introduce a software reasoning framework that integrates the automation of induction with first-order theorem proving. Our framework allows us to automatically discharge verification conditions of sorting/permutation programs, without requiring manually proven or a priori given loop invariants. In particular, we automatically derive induction axioms to establish the functional correctness of the recursive implementation of Quicksort from Figure 1. In a nutshell, we proceed as follows.

(i) We formalize the *semantics of functional programs* in extensions of the first-order theory of lists (Section 3). Rather than focusing on lists with a specific background theory, such as integers/naturals, our formalization relies on a parameterized sort/type a abstracting specific (arithmetic) theories. To this end, we impose that the sort a has a linear order $\leq$. We then express program semantics in the first-order theory of lists parameterized by a, allowing us to quantify over lists of sort a as they are domain elements of our first-order theory.

(ii) We revise inductive reasoning in first-order theorem proving (Section 4) and introduce *computation induction* as a means to tackle recursive sorting algorithms. We, therefore, extend the first-order reasoner with an inductive infer-

ence based on the computation induction schema and outline its necessity for recursive sorting routines.
(iii) We leverage *first-order theorem proving for compositional proofs* of recursive parameterized sorting algorithms (Section 5), in particular of `Quicksort` from Figure 1. Our proofs do not rely on manually proven invariants or other forms of inductive annotations. Rather, we embed the application of induction directly in saturation proving and manually split (sorting) verification conditions into multiple proof obligations when necessary. Each such condition represents a first-order lemma, and hence a proof step, that is proved by saturation with induction. Specifically, all our lemmas/verification conditions are automatically proven by means of structural and/or computation induction during the saturation process. Thanks to the automation of induction in saturation, we turn first-order theorem proving into a powerful approach to guide human reasoning about recursive properties. We do not rely on user-provided inductive properties, but generate inductive hypotheses/invariants via inductive inferences as logical consequences of our program semantics.
(iv) We note that sorting algorithms often follow a divide-and-conquer approach (see Figure 2). We show how proof search can be guided via compositional proof splitting for such routines, and provide a *generalized set of lemmas* that is applicable to functional sorting algorithms on recursive data structures, such as lists (Section 6). Doing so, we remark that one of the major reasoning burdens towards establishing the correctness of sorting algorithms comes with formalizing permutation properties, for example that two lists are permutations of each other. Universally quantifying over permutations of lists is, however, not a first-order property and hence reasoning about list permutation requires higher-order logic. While counting and comparing the number of list elements is a viable option to formalize permutation equivalence in first-order logic, the necessary arithmetic reasoning adds an additional burden to the underlying prover. We overcome this challenge by introducing an effective first-order formalization of *permutation equivalence* over parameterized lists. Our permutation equivalence property encodes *multiset* operations over lists, eliminating the need of counting list elements, and therefore arithmetic reasoning, or fully axiomatizing (higher-order) permutations.

Contributions. In summary, we bring the following main contributions.

(i) We introduce the formal foundations for formalizing the semantics of functional programs with recursive data structures in the first-order theory of lists with parameterized sorts. Doing so, we capture the correctness of sorting routines via two properties over lists, namely the sortedness property and the permutation equivalence property, and introduce a first-order formalization of these properties (Section 3).

(ii) We extend first-order theorem proving to include inductive inferences based on computation induction, enabling automated inductive reasoning with first-order provers over recursive functions (Section 4).

(iii) We showcase compositional reasoning via first-order theorem provers with built-in induction and provide a fully automated compositional correctness proof of the recursive `Quicksort` algorithm of Figure 1 (Section 5). We emphasize that the only manual effort in our framework comes with splitting formulas into multiple lemmas (Section 6.1); each lemma is established automatically by means of automated theorem proving with built-in induction.

(iv) We generalize our inductive lemmas to prove correctness of multiple functional sorting algorithms (Section 6.2), including `Mergesort` and `Insertionsort`.

(v) We demonstrate our findings (Section 7) by implementing our approach on top of the VAMPIRE theorem prover [12], providing thus a fully automatd tool support towards validating the functional correctness of sorting algorithms.

2 Preliminaries

We assume familiarity with standard first-order logic (FOL) and briefly introduce saturation-based proof search in first-order theorem proving [12].

Saturation. Rather than using arbitrary first-order formulae G, most first-order theorem provers rely on a clausal representation C of G. The task of first-order theorem proving is to establish that a formula/goal G is a logical consequence of a set $\mathcal{A}$ of clauses, including assumptions. Doing so, first-order provers clausify the negation $\neg G$ of G and derive that the set $S = \mathcal{A} \cup \{\neg G\}$ is unsatisfiable[1]. To this end, first-order provers *saturate* S by computing all logical consequences of S with respect to some sound inference system $\mathcal{I}$. A sound inference system $\mathcal{I}$ derives a clause D from clauses C such that $C \rightarrow D$. The saturated set of S w.r.t. $\mathcal{I}$ is called the *closure* of S w.r.t. $\mathcal{I}$, whereas the process of deriving the closure of S is called *saturation*. By soundness of $\mathcal{I}$, if the closure of S contains the empty clause $\square$, the original set S of clauses is unsatisfiable, implying the validity of $\mathcal{A} \rightarrow G$; in this case, we established a *refutation* of $\neg G$ from $\mathcal{A}$, hence a proof of validity of G.

The *superposition calculus* is a common inference system used by saturation-based provers for FOL with equality [18]. The superposition calculus is sound and *refutationally complete*: for any unsatisfiable formula $\neg G$, superposition-based saturation derives the empty clause $\square$ as a logical consequence of $\neg G$.

Parameterized Lists. We use the first-order theory of recursively defined datatypes [11]. In particular, we consider the list datatype with two constructors nil and $\mathsf{cons}(x, xs)$, where nil is the empty list and x and xs are respectively the head and tail of a list. We introduce a type parameter a that abstracts the sort/background theory of the list elements. Here, we impose the restriction that the sort a has a linear order $<$, that is, a binary relation which is reflexive, antisymmetric, transitive and total. For simplicity, we also use $\geq$ and $\leq$ as the standard ordering extensions of $<$. As a result, we work in the first-order theory of lists parameterized by sort a, allowing us to quantify over lists as domain elements of this theory. For simplicity, we write xs_a, ys_a, zs_a to mean that the lists xs, ys, zs are parameterized by sort a; that is their elements are of sort a. Similarly, we use x_a, y_a, z_a to mean that the list elements x, y, z are of sort a. Whenever it is clear from the context, we omit specifying the sort a.

[1] for simplicity, we denote by $\neg G$ the clausified form of the negation of G

Function definitions. We make the following abuse of notation. For some function `f` in some program `P`, we use the notation `f(arg`$_1$`, ...)` to refer to function definitions/calls appearing in the input algorithm, while the mathematical notation $f(arg_1, ...)$ refers to its pendant in our logical representation, that is the function call semantics in first-order notation as introduced in Section 3.

3 First-Order Semantics of Functional Sorting Algorithms

We outline our formalization of recursive sorting algorithms in the full first-order theory of parameterized lists.

3.1 Recursive Functions in First-Order Logic

We investigate recursive algorithms written in a functional coding style and defined over lists using list constructors. That is, we consider recursive functions `f` that manipulate the empty list nil and/or the list $\mathsf{cons}(x, xs)$.
Many recursive sorting algorithms, as well as other recursive operations over lists, implement a *divide-and-conquer* approach: let `f` be a function following such a pattern, `f` uses (i) a *partition function* to divide $list_a$, that is a *list* of sort a, into two smaller sublists upon which `f` is recursively applied to, and (ii) calls a *combination function* that puts together the result of the recursive calls of `f`. Figure 2 shows such a divide-and-conquer pattern, where the partition function `partition` uses an invertible operator $\circ$, with $\circ^{-1}$ being the complement of $\circ$; `f` is applied to the results of $\circ$ before these results are merged using the combination function `combine`.

Note that the recursive function `f` of Figure 2 is defined via the declaration f :: $a'list$ $\rightarrow$ ... $\rightarrow$ $a'list$, where ... denotes further input parameters. We formalize the first-order semantics of `f` via the function $f\colon (list_a \times ...) \mapsto list_a$, by translating the inductive function definitions `f` to the following first-order formulas with parameterized lists (in first-order logic, function definitions can be considered as universally quantified equalities):

```
f :: a' list → ... → a' list
f(nil, ...) = nil
f(cons(y,ys), ...) =
   combine(
      f(partition∘(cons(y,ys))),
      f(partition∘^-1(cons(y,ys)))
   )

```

Fig. 2: Recursive divide-and-conquer approach.

$$\begin{aligned} f(\mathsf{nil}) &= \mathsf{nil} \\ \forall x_a, xs_a.\ f(\mathsf{cons}(x, xs)) &= combine(\ f(partition_{\circ}(\mathsf{cons}(x, xs))), \\ &\qquad\qquad\qquad f(partition_{\circ^{-1}}(\mathsf{cons}(x, xs)))). \end{aligned} \tag{1}$$

The recursive divide-and-conquer pattern of Figure 2, together with the first-order semantics (1) of `f`, is used in Sections 5-6 for proving correctness of the

`Quicksort` algorithm (and other sorting algorithms), as well as for applying lemma generalizations for divide-and-conquer list operations. We next introduce our first-order formalization for specifying that `f` implements a sorting routine.

3.2 First-Order Specification of Sorting Algorithms

We consider a specific function instance of `f` implementing a sorting algorithm, expressed through $sort :: a'list \rightarrow a'list$. The functional behavior of *sort* needs to satisfy two specifications implying the functional correctness of *sort*: (i) sortedness and (ii) permutations equivalence of the list computed by *sort*.
(i) Sortedness: *The list computed by the sort function must be sorted w.r.t. some linear order* $\leq$ *over the type* a *of list elements.* We define a parameterized version of this sortedness property using an inductive predicate *sorted* as follows:

$$\begin{gathered} sorted(\mathsf{nil}) = \top \\ \forall x_a, xs_a .\, sorted(\mathsf{cons}(x, xs)) = (elem_{\leq}list(x, xs) \wedge sorted(xs)), \end{gathered} \tag{2}$$

where $elem_{\leq}list(x, xs)$ specifies that $x \leq y$ for any element y in xs. Proving correctness of a sorting algorithm *sort* thus reduces to proving the validity of:

$$\forall xs_a .\, sorted(sort(xs)). \tag{3}$$

(ii) Permutation Equivalence: *The list computed by the sort function is a permutation of the input list to the sort function.* In other words the input and output lists of *sort* are permutations of each other, in short permutation equivalent.
Axiomatizing permutations requires quantification over relations and is thus not expressible in first-order logic [14]. A common approach to prove permutation equivalence of two lists is to count the occurrence of elements in each list respectively and compare the occurrences of each element. Yet, counting adds a burden of arithmetic reasoning over naturals to the underlying prover, calling for additional applications of mathematical induction. We overcome these challenges of expressing permutation equivalence as follows. We introduce a family of functions $filter_Q$ manipulating lists, with the function $filter_Q$ being parameterized by a predicate Q and as given in Figure 3.

```
filter_Q :: a' → a' list → a' list
filter_Q(x, nil) = nil
filter_Q(x, cons(y, ys))=
   if (Q(y, x)){
      cons(y, filter_Q(x, ys))
   } else {
      filter_Q(x, ys)
   }
```

Fig. 3: Functions $filter_Q$ filtering elements of a list, by using a predicate $Q(y, x)$ over list elements x, y.

In particular, given an element x and a list ys, the functions $filter_=$, $filter_<$, and $filter_\geq$ compute the maximal sublists of ys that contain only equal, resp. smaller and greater-or-equal elements to x. Analogously to counting the multiset multiplicity of x in ys via counting functions, we compare lists given by $filter_=$, avoiding the need to count the number of occurrences of x and hence prolific axiomatizations of arithmetic. Thus, to prove that the input/output lists of *sort* are permutation equivalent, we show that, for every list element x, the results of applying `filter`$_=$ to the input/output list of *sort* are the same over all elements. Formally, we have the following first-order property of permutation equivalence:

$$\forall x_a, xs_a\,.\; filter_=(x, xs) = filter_=(x, sort(xs)). \tag{4}$$

4 Computation Induction in Saturation

In this section, we describe our reasoning extension to saturation-based first-order theorem proving, in order to support inductive reasoning for recursive sorting algorithms as introduced in Section 3. Our key reasoning ingredient comes with a structural induction schema of *computation induction*, which we directly integrate in the saturation proving process.
Inductive reasoning has recently been embedded in saturation-based theorem proving [7], by extending the superposition calculus with a new inference rule based on *induction axioms*:

$$(\mathsf{Ind})\ \frac{\overline{L}[t] \vee C}{\mathsf{cnf}(\neg F \vee C)} \quad \text{where} \quad \begin{array}{l} (1)\ L[t] \text{ is a quantifier-free (ground) literal,} \\ (2)\ F \to \forall x.L[x] \text{ is a valid } \textit{induction axiom}, \\ (3)\ \mathsf{cnf}(\neg F \vee C) \text{ is the clausal form of } \neg F \vee C. \end{array}$$

An *induction axiom* refers to an instance of a valid induction schema. In our work, we use structural and computational induction schemata.
In particular, we use the following *structural induction* schema over lists:

$$\big(F[\mathsf{nil}] \wedge \forall x, ys.(F[ys] \to F[\mathsf{cons}(x, ys)])\big) \to \forall zs.F[zs] \tag{5}$$

Then, considering the induction axiom resulting from applying schema (5) to L, we obtain the following Ind instance for lists:

$$\frac{\overline{L}[t] \vee C}{\begin{array}{c} \overline{L}[\mathsf{nil}] \vee L[\sigma_{ys}] \vee C \\ \overline{L}[\mathsf{nil}] \vee \overline{L}[\mathsf{cons}(\sigma_x, \sigma_{ys})] \vee C \end{array}}$$

where t is a ground term of sort list, $L[t]$ is ground, and σ_x and σ_{ys} are fresh constant symbols. The above Ind instance yields two clauses as conclusions and is applied during the saturation process.

Sorting algorithms, however, often require induction axioms that are more complex than instances of structural induction (5). Such axioms are typically instances of the computation/recursion induction schema, arising from divide-and-conquer strategies as introduced in Section 3.1. Particularly, the complexity arises due to the two recursive calls on different parts of the original input list produced by the *partition* function that have to be taken into account by the induction schema. We therefore use the following *computation induction* schema over lists:

$$\left(F[\mathsf{nil}] \land \forall x, ys.\left(\left(\begin{matrix}F[partition_\circ(x, ys)]\land\\ F[partition_{\circ^{-1}}(x, ys)]\end{matrix}\right) \rightarrow F[\mathsf{cons}(x, ys)]\right)\right) \rightarrow \forall zs.F[zs] \quad (6)$$

yielding the following instance of Ind that is applied during saturation:

$$\frac{\overline{L}[t] \lor C}{\begin{matrix}\overline{L}[\mathsf{nil}] \lor L[partition_\circ(\sigma_x, \sigma_{ys})] \lor C\\ \overline{L}[\mathsf{nil}] \lor L[partition_{\circ^{-1}}(\sigma_x, \sigma_{ys})] \lor C\\ \overline{L}[\mathsf{nil}] \lor \overline{L}[\mathsf{cons}(\sigma_x, \sigma_{ys})] \lor C\end{matrix}}$$

where t is a ground term of sort list, $L[t]$ is ground, σ_x and σ_{ys} are fresh constant symbols, and $partition_\circ$ and its complement refer to the functions that partition lists into sublists within the actual sorting algorithms.

5 Proving Recursive `Quicksort`

We now describe our approach towards proving the correctness of the recursive parameterized version of `Quicksort`, as given in Figure 1. Note that `Quicksort` recursively sorts two sublists that contain respectively smaller and greater-or-equal elements than the pivot element x of its input list. We reduce the task of proving the functional correctness of `Quicksort` to the task of proving the (i) sortedness property (3) and (ii) the permutation equivalence property (4) of `Quicksort`. As mentioned in Section 3.2, a similar reasoning is needed for most sorting algorithms, which we evidence in Sections 6–7.

5.1 Proving Sortedness for `Quicksort`

Given an input list xs, we prove that `Quicksort` computes a sorted list by considering the property (3) instantiated for `Quicksort`. That is, we prove:

$$\forall xs_a.\ sorted(quicksort(xs)) \quad (7)$$

The sortedness property (7) of `Quicksort` is proved via *compositional reasoning* over (7). Namely, we enforce the following two properties that together imply (7):

(S1) By using the linear order $\leq$ of the background theory a, for any element y in the sorted list $quicksort(filter_<(x, xs))$ and any element z in the sorted list $quicksort(filter_\geq(x, xs))$, we have $y \leq x \leq z$.

(S2) The functions $filter_<$ and $filter_\geq$ of Figure 3 are correct. That is, filtering elements from a list that are smaller, respectively greater-or-equal, than an element x results in sublists only containing elements smaller than, respectively greater-or-equal, than x.

Similarly to (2) and to express property **(S2)**, we introduce the inductively defined predicates $elem_{\leq}list :: a' \to a'list \to bool$ and $list_{\leq}list :: a'list \to a'list \to bool$:

$$\begin{aligned} \forall x_a .\, elem_{\leq}list(x, \mathsf{nil}) &= \top \\ \forall x_a, y_a, ys_a .\, elem_{\leq}list(x, \mathsf{cons}(y, ys)) &= x \leq y \land elem_{\leq}list(x, ys), \end{aligned} \tag{8}$$

and

$$\begin{aligned} \forall ys_a .\, list_{\leq}list(\mathsf{nil}, ys) &= \top \\ \forall x_a, xs_a, ys_a .\, list_{\leq}list(\mathsf{cons}(x, xs), ys) &= (elem_{\leq}list(x, ys) \land list_{\leq}list(xs, ys)). \end{aligned} \tag{9}$$

Thus, for some element x and lists xs, ys, we express that x is smaller than or equal to any element of xs by $elem_{\leq}list(x, xs)$. Similarly, $list_{\leq}list(xs, ys)$ states that every element in list xs is smaller than or equal to any element in ys.
The inductively defined predicates of (8)–(9) allow us to express necessary lemmas over list operations preserving the sortedness property (7), for example, to prove that appending sorted lists yields a sorted list.
Proving properties **(S1)**–**(S2)**, and hence deriving the sortedness property (7) of `Quicksort`, requires *three first-order lemmas* in addition to the first-order semantics (1) of `Quicksort`. Each of these lemmas is automatically proven by saturation-based theorem proving using the structural and/or computation induction schemata of (5) and (6); hence, by compositionality, we obtain **(S1)**–**(S2)** implying (7). We next discuss these three lemmas and outline the complete (compositional) proof of the sortedness property (7) of `Quicksort`.
• In support of **(S1)**, lemma (10) expresses that for two *sorted* lists xs, ys and a list element x, such that $elem_{\leq}list(x, xs)$ holds and all elements of the constructed list $\mathsf{cons}(x, xs)$ are greater than or equal to all elements in ys, the result of concatenating ys and $\mathsf{cons}(x, xs)$ yields a sorted list. Formally, we have

$$\begin{aligned} \forall x_a, xs_a, ys_a .\; &\big(sorted(xs) \land sorted(ys) \land elem_{\leq}list(x, xs) \land \\ &\; list_{\leq}list(ys, \mathsf{cons}(x, xs))\big) \\ &\to sorted(append(ys, \mathsf{cons}(x, xs))) \end{aligned} \tag{10}$$

• In support of **(S2)**, we need to establish that filtering greater-or-equal elements for some list element x results in a list whose elements are greater-or-equal than x. In other words, the inductive predicate of (8) is invariant over sorting and filtering operations over lists.

$$\forall x_a, xs_a .\, elem_{\leq}list(x, quicksort(filter_{\geq}(x, xs))). \tag{11}$$

• Lastly and in further support of **(S1)**–**(S2)**, we establish that all elements of a list xs are "covered" with the filtering operations $\texttt{filter}_{\geq}$ and $\texttt{filter}_{<}$ w.r.t. a list element x of xs. Intuitively, a call of $\texttt{filter}_{<}\texttt{(x,xs)}$ results in a list containing all elements of xs that are smaller than x, while the remaining elements of xs are those that are greater-or-equal than x and hence are contained in $\mathsf{cons}(x, filter_{\geq}(x, xs))$. By applying `Quicksort` over the input list xs, we get:

$$\begin{aligned}&\forall x_a, xs_a \,.\, list_{\leq}list(\\ &\quad quicksort(filter_{<}(x, xs)), \\ &\quad \mathsf{cons}(x, quicksort(filter_{\geq}(x, xs)))).\end{aligned} \tag{12}$$

The first-order lemmas (10)–(12) guide saturation-based proving to instantiate structural/computation induction schemata and derive the following induction axiom necessary to prove **(S1)**–**(S2)**, and hence sortedness of `Quicksort`:

$$\begin{aligned}&\Big(sorted(quicksort(\mathsf{nil}))\wedge \\ &\quad \forall x_a, xs_a \,.\, \begin{pmatrix} sorted(quicksort(filter_{\geq}(x, xs)))\wedge \\ sorted(quicksort(filter_{<}(x, xs)))\end{pmatrix} \to sorted(quicksort(\mathsf{cons}(x, xs)))\Big) \\ &\to \forall xs_a \,.\, sorted(quicksort(xs)),\end{aligned} \tag{13}$$

where axiom (13) is automatically obtained during saturation from the computation induction schema (6). Intuitively, the prover replaces F by $sorted(quicksort(t))$ for some term t, and uses $filter_{<}$ and $filter_{\geq}$ as $partition_{\circ}$ and $partition_{\circ^{-1}}$ respectively to find the necessary computation induction schema. We emphasize that this step is fully automated during the saturation run.
The first-order lemmas (10)–(12), together with the induction axiom (13) and the first-order semantics (1) of `Quicksort`, imply the sortedness property (4) of `Quicksort`; this proof can automatically be derived using saturation-based reasoning. Yet, the obtained proof assumes the validity of each of the lemmas (10)–(12). To eliminate this assumption, we propose to also prove lemmas (10)–(12) via saturation-based reasoning. Yet, while lemma (10) is established by saturation with structural induction (5) over lists, proving lemmas (11)–(12) requires further first-order formulas. In particular, for proving lemmas (11)–(12) via saturation, we use four further lemmas, as follows.
• Lemmas (14)–(15) indicate that the order of $elem_{\leq}list$ and $list_{\leq}list$ is preserved under $quicksort$, respectively. That is,

$$\forall x_a, xs_a \,.\, elem_{\leq}list(x, xs) \to elem_{\leq}list(x, quicksort(xs)) \tag{14}$$

and

$$\forall xs_a, ys_a \,.\, list_{\leq}list(ys, xs) \to list_{\leq}list(quicksort(ys), xs). \tag{15}$$

• Proving lemmas (14)–(15), however, requires two further lemmas that follow from saturation with built-in computation and structural induction, respectively. Namely, lemmas (16)–(17) establish that $elem_{\leq}list$ and $list_{\leq}list$ are also invariant over appending lists. That is,

$$\begin{aligned}\forall x_a, y_a, xs_a, ys_a \,.\, &\big(y \leq x \wedge elem_{\leq}list(y, xs) \wedge elem_{\leq}list(y, ys)\big) \\ &\to elem_{\leq}list(y, append(\mathsf{cons}(x, ys), xs))\end{aligned} \tag{16}$$

and

$$\begin{aligned}\forall xs_a, ys_a, zs_a \,.\, &\big(list_{\leq}list(ys, xs) \wedge list_{\leq}list(zs, xs)\big) \\ &\to list_{\leq}list(append(ys, zs), xs)\end{aligned} \tag{17}$$

With lemmas (14)–(17), we automatically prove lemmas (10)–(12) via saturation-based reasoning. The complete automation of proving properties **(S1)**-**(S2)**, and hence deriving the sortedness property (7) of `Quicksort` in a compositional manner, requires thus *altogether seven lemmas* in addition to the first-order semantics (1) of `Quicksort`. *Each of these lemmas is automatically established via saturation with built-in induction.* Hence, unlike interactive theorem proving, compositional proving with first-order theorem provers can be leveraged to eliminate the need to a priori specifying necessary induction axioms.

5.2 Proving Permutation Equivalence for `Quicksort`

In addition to establishing the sortedness property (7) of `Quicksort`, the functional correctness of `Quicksort` also requires proving the permutation equivalence property (4) for `Quicksort`. That is, we prove:

$$\forall x_a, xs_a \,.\, filter_{=}(x, xs) = filter_{=}(x, quicksort(xs)). \tag{18}$$

In this respect, we follow the approach introduced in Section 3.2 to enable first-order reasoning over permutation equivalence (18). Namely, we use $filter_{=}$ to filter elements x in the lists xs and $quicksort(xs)$, respectively, and build the corresponding multisets containing as many x as x occurs in xs and $quicksort(xs)$. By comparing the resulting multisets, we implicitly reason about the number of occurrences of x in xs and $quicksort(xs)$, yet, without the need to explicitly count occurrences of x. In summary, we reduce the task of proving (18) to *compositional reasoning* again, namely to proving following *two properties given as first-order lemmas* which, by compositionality, imply (18):
(P1) List concatenation commutes with $filter_{=}$, expressed by the lemma:

$$\forall x_a, xs_a, ys_a \,.\, filter_{=}(x, append(xs, ys)) = append(\, filter_{=}(x, xs), \; filter_{=}(x, ys)). \tag{19}$$

(P2) Appending the aggregate of both `filter`-operations results in the same multisets as the unfiltered list, that is, permutation equivalence is invariant over combining complementary reduction operations. This property is expressed via:

$$\forall x_a, y_a, xs_a \,.\, filter_{=}(x, xs) = append(\, filter_{=}(x, filter_{<}(y, xs)), \; filter_{=}(x, filter_{\geq}(y, xs))). \tag{20}$$

Similarly as in Section 5.1, we prove lemmas (**(P1)**)–(**(P2)**) by saturation-based reasoning with built-in induction. In particular, investigating the proof output shows that lemma (**(P1)**) is established using the structural induction schema (5) in saturation, while the validity of lemma (**(P2)**) is obtained by applying the computation induction schema (6) in saturation.
By proving lemmas (**(P1)**)–(**(P2)**), we thus establish validity of permutation equivalence (18) for `Quicksort`. Together with the sortedness property (7) of `Quicksort` proven in Section 5.1, we conclude the functional correctness of `Quicksort` in a fully automated and compositional manner, using saturation-based theorem proving with built-in induction and *altogether nine first-order lemmas* in addition to the first-order semantics (1) of `Quicksort`.

6 Lemma Generalizations for Guided Proof Splits

Establishing the functional correctness of `Quicksort` in Section 5 uses nine first-order lemmas that express inductive properties over lists in addition to the first-order semantics (1) of `Quicksort`. While each of these lemmas is proved by saturation using structural/computation induction schemata, coming up with proper inductive lemmas remains crucial in reasoning about inductive data structures. That is, we need effective ways to split the proof so that the first-order theorem prover can automatically discharge all proof steps with built-in induction.
In Section 6.1, we describe when and how we split proof obligations into lemmas, so that each of these lemmas can further be proved automatically using first-order theorem proving. In Section 6.2, we next demonstrate that the lemmas of Section 5 can be generalized and leveraged to prove correctness of other divide-and-conquer list sorting algorithms, in particular within the `Mergesort` routine of Figure 5. The generality of our inductive lemmas from Section 5 also helps reasoning about sorting routines that do not necessarily follow a divide-and-conquer strategy, such as the `Insertionsort` algorithm of (Figure 4).

6.1 Guided Proof Splitting

Contrary to automated approaches that use inductive annotations to alleviate inductive reasoning, our approach synthesizes the correct induction axioms automatically during saturation runs. However, there is still a manual limitation to our approach, namely proof splitting. That is, deciding when a lemma is necessary or helpful for the automated reasoner.
Splitting the proof into multiple lemmas is necessary to guide the prover to find the right terms to apply the inductive inferences of Section 4. This is particularly the case when input problems, such as the sorting algorithms, contain calls to multiple recursive functions - each of which has to be shown to preserve the property that is to be verified.
We next illustrate and examine the need for proof splitting using lemma (10).

Example 1 (Compositional reasoning over sortedness in saturation). Consider the following stronger version of lemma (10) in the proof of `Quicksort`:

$$\begin{aligned}&\forall x_a, xs_a, ys_a \,.\\ &\quad \big(sorted(xs) \land sorted(ys)\big) \to sorted(append(ys, \mathsf{cons}(x, xs))).\end{aligned} \tag{21}$$

This formula could automatically be derived by saturation with computation induction (6) while trying to prove sortedness of the algorithm. However, formula (21) is not helpful with regards to the specification of `Quicksort` since the value of x is not correctly restricted w.r.t. $\leq$ to xs, ys (e.g. concatenating a sorted xs with an arbitrary x not necessarily yields a sorted list). The prover needs additional information to verify sortedness. Therefore, the assumptions $elem_{\leq}list(x, xs)$ and $list_{\leq}list(ys, \mathsf{cons}(x, xs))$ are needed in addition to (21), resulting in lemma (10). Yet, lemma (10) from Section 5 can be automatically

```
insertsort :: a' list → a' list
insertsort(nil) = nil
insertsort(cons(x, xs)) = insert(x, insertsort(xs))

insert :: a' → a' list → a' list
insert(x, nil) = cons(x, nil)
insert(x, cons(y, ys)) =
  if (x ≤ y) {
    cons(x, cons(y, ys))
  } else {
    cons(y, insert(x, ys))
  }

```

Fig. 4: Recursive algorithm of `Insertionsort` using the recursive function definition `insertsort` and auxiliary (recursive) function `insert`. `Insertionsort` recursively sorts the list by inserting single elements in the correct order with the helper function `insert`.

derived via saturation with *compositional reasoning*, based on computation induction (6). That is, we manually split proof obligations based on missing information in the saturation runs: we derive (21) from (6) via saturation, strengthen the hypotheses of (21) with missing necessary conditions $elem_{\leq}list(x, xs)$ and $list_{\leq}list(ys, \mathsf{cons}(x, xs))$, and prove their validity via saturation, yielding (10).

Manual Formula Splits for Automated Proofs. Contrary to loop invariants or other inductive annotations that are rarely proven correct by means of the underlying verification technique itself, our approach automatically proves each lemma correct by synthesizing the correct induction axioms during proof search. In case a proof fails, we investigate and manually strengthen the synthesized induction axioms and verify their validity in turn again with the theorem prover and built-in induction. That is, we do not simply assume inductive lemmas but also provide a formal argument of their validity. We emphasize that we manually split the proof into multiple verification conditions such that inductive reasoning can fully be automated in saturation.

6.2 Lemma Generalizations for Sorting

The lemmas from Section 5 represent a number of common proof splits that can be applied to various list sorting tasks. In the following we generalize their structure and apply them to two other sorting algorithms, namely `Mergesort` and `Insertionsort`.

Common Patterns of Inductive Lemmas for Sorting Algorithms. Consider the computation induction schema (6). When using (6) for proving the sortedness (7) and permutation equivalence (18) of `Quicksort`, the inductive formula F of (6) is, respectively, instantiated with the predicates *sorted* from (7) and $filter_{=}$ from (18). The base case $F[\mathsf{nil}]$ of schema (6) is then trivially proved by saturation for both properties (7) and (18) of `Quicksort`.

```
mergesort :: a' list → a' list
mergesort(nil) = nil
mergesort(xs) =
  merge(
    mergesort(take((xs_length div 2), xs)) ,
    mergesort(drop((xs_length div 2), xs))
  )

merge :: a' list → a' list → a' list
merge(nil, ys) = ys
merge(xs, nil) = xs
merge(cons(x, xs), cons(y, ys)) =
  if (x ≤ y) {
    cons(x, merge(xs, cons(y, ys)))
  } else {
    cons(y, merge(cons(x, xs), ys))
  }

```

Fig. 5: Recursive `Mergesort` using the recursive functions `merge`, `take`, and `drop` over lists of sort a. `Mergesort` splits the input list xs into two halves by using `take` and `drop` that respectively *take* and *drop* the first half of elements of the input list (corresponding to `partition` functions of Figure 2). Both halves are recursively sorted and combined by the `merge` function, yielding a sorted list (corresponding to `combine` of Figure 2).

Proving the induction step case of schema (6) is however challenging as it relies on *partition*-functions which are further used by *combine* functions within the divide-and-conquer patterns of Figure 2. Intuitively this means, that proving the induction step case of schema (6) for the sortedness (7) and permutation equivalence (18) properties requires showing that applying *combine* functions over *partition* functions preserve sortedness (7) and permutation equivalence (18), respectively. For divide-and-conquer algorithms of Figure 2, the step case of schema (6) requires thus proving the following lemma:

$$\left(\forall x_a, ys_a.\left(combine\begin{pmatrix}F[partition_\circ(x,ys)]\wedge\\ F[partition_{\circ^{-1}}(x,ys)]\end{pmatrix}\rightarrow F[\mathsf{cons}(x,ys)]\right)\right). \tag{22}$$

We next describe generic instances of lemma (22) to be used within proving functional correctness of sorting algorithms, depending on the *partition*/*combine* function of the underlining divide-and-conquer sorting routine.

(i) ***Combining sorted lists preserves sortedness.*** For proving the inductive step case (22) of the sortedness property (3) of sorting algorithms, we require the following generic lemma (23):

$$\forall xs_a, ys_a.\,\big(sorted(xs)\wedge sorted(ys)\big)\rightarrow sorted(combine(xs,ys)), \tag{23}$$

ensuring that combining sorted lists results in a sorted list. Lemma (23) is used to establish property **(S1)** of `Quicksort`, namely used as lemma (10) for proving the preservation of sortedness under the *append* function.
We showcase that generality of lemma (23), by using it upon sorting routines different than `Quicksort`. Consider, for example, `Mergesort` as given in Figure 5. The sortedness property (3) of `Mergesort` can be proved by using saturation with lemma 23; note that the `merge` function of `Mergesort` acts as a *combine* function of (23). That is, we establish the sortedness property of `Mergesort` via the following instance of (23):

$$\forall xs_a, ys_a \,.\, sorted(xs) \wedge sorted(ys) \rightarrow sorted(merge(xs, ys)) \tag{24}$$

Finally, lemma (23) is not restricted to divide-and-conquer routines. For example, when proving the sortedness property (3) of the `Insertionsort` algorithm of Figure 4, we use saturation with lemma (23) applied to `insert`. As such, sortedness of `Insertionsort` is established by the following instance of (23):

$$\forall x_a, xs_a \,.\, sorted(xs) \rightarrow sorted(insert(x, xs)) \tag{25}$$

(ii) ***Combining reductions preserves permutation equivalence.*** Similarly to Section 5.2, proving permutation equivalence (4) over divide-and-conquer sorting algorithms of Figure 2 is established via the following two properties:
• As in **(P1)** for `Quicksort`, we require that *combine* commutes with $filter_{=}$:

$$\forall x_a, xs_a, ys_a \,.\, filter_{=}(x, combine(xs, ys)) = combine(filter_{=}(x, xs), \\ filter_{=}(x, ys)) \tag{26}$$

• Similarly to **(P2)** for `Quicksort`, we ensure that, by combining (complementary) *reduction* functions, we preserve (4). That is,

$$\forall x_a, xs_a \,.\, filter_{=}(x, xs) = combine(filter_{=}(x, partition_{\circ}(xs)), \\ filter_{=}(x, partition_{\circ^{-1}}(xs))) \tag{27}$$

Note that lemmas **(P1)** and **(P2)** for `Quicksort` are instances of (26) and (27) respectively, as the *append* function of `Quicksort` acts as a *combine* function and the $filter_{<}$ and $filter_{\geq}$ functions are the *partition* functions of Figure 2.
To prove the permutation equivalence (4) property of `Mergesort`, we use the functions `take` and `drop` as the *partition* functions of lemmas (26)–(27). Doing so, we embed a natural number argument n in lemmas (26)–(27), with n controlling how many list elements are *taken* and *dropped*, respectively, in `Mergesort`. As such, the following instances of lemmas (26)–(27) are adjusted to `Mergesort`:

$$\forall x_a, xs_a, ys_a \,.\, filter_{=}(x, merge(xs, ys)) = append(filter_{=}(x, xs), \\ filter_{=}(x, ys)) \tag{28}$$

and

$$\forall x_a, n_{\mathbb{N}}, xs_a \,.\, filter_{=}(x, xs) = append(filter_{=}(x, take(n, xs)), \\ filter_{=}(x, drop(n, xs))), \tag{29}$$

with lemmas (28)–(29) being proved via saturation. With these lemmas at hand, the permutation equivalence (4) of `Mergesort` is established, similarly to `Quicksort`.

Finally, the generality of lemmas (26)–(27) naturally pays off when proving the permutation equivalence property (4) of `Insertionsort`. Here, we only use a simplified instance of (26) to prove (4) is preserved by the auxiliary function `insert`. That is, we use the following instance of (26):

$$\forall x_a, y_a, ys_a.\, filter_{=}(x, \mathsf{cons}(y, ys)) = filter_{=}(x, insert(y, ys)), \qquad (30)$$

which is automatically derivable by saturation with computation induction (6). We conclude by emphasizing the generality of the lemmas (23) and (26)–(27) for automating inductive reasoning over sorting algorithms in saturation-based first-order theorem proving: functional correctness of `Quicksort`, `Mergesort`, and `Insertionsort` are proved using these lemmas in saturation with induction. Moreover, each of these lemmas is established via saturation with induction. Thus, compositional reasoning in saturation with computation induction enables proving challenging sorting algorithms in a fully automated manner.

7 Implementation and Experiments

Implementation. Our work on saturation with induction in the first-order theory of parameterized lists is implemented in the first-order prover VAMPIRE [12]. In support of parameterization, we extended the SMT-LIB parser of VAMPIRE to support parametric data types from SMT-LIB [1] – version 2.6. In particular, using the `par` keyword, our parser interprets `(par (a`$_1$ `...` `a`$_n$`) ...)` similar to universally quantified blocks where each variable $\mathtt{a}_i$ is a type parameter.
Appropriating a generic saturation strategy, we adjust the simplification orderings (LPO) for efficient equality reasoning/rewrites to our setting. For example, the precedence of function *quicksort* is higher than of symbols nil, cons, *append*, $filter_{<}$ and $filter_{\geq}$, ensuring that *quicksort* function terms are expanded to their functional definitions.
We further apply recent results of encompassment demodulation [3] to improve equality reasoning within saturation (`-drc encompass`). We use induction on data types (`-ind struct`), including complex data type terms (`-indoct on`).

Experimental Evaluation. We evaluated our approach over challenging recursive sorting algorithms taken from [17], namely `Quicksort`, `Mergesort`, and `Insertionsort`. We show that the functional correctness of these sorting routines can be verified automatically by means of saturation-based theorem proving with induction, as summarized in Table 1.
We divide our experiments according to the specification of sorting algorithms: the first column `PermEq` shows the experiments of all sorting routines w.r.t. permutation equivalence (4), while `Sortedness` refers to the sortedness (3) property, together implying the functional correctness of the respective sorting algorithm. Here, the inductive lemmas of Sections 5–6 are proven in separate saturation

PermEq			
Benchm.	Pr.	T	Required lemmas
IS-PE	✓	0.02	{IS-PE-L1}
IS-PE-L1	✓	0.13	∅
MS-PE	✓	0.06	{MS-PE-L1, MS-PE-L2}
MS-PE-L1	✓*	0	-
MS-PE-L2	✓	0.03	∅
MS-PE-L3	✓	0.15	∅
QS-PE	✓	0.5	{QS-PE-L1, QS-PE-L2}
QS-PE-L1	✓	0.05	∅
QS-PE-L2	✓	0.09	∅

Table 1: Experimental evaluation of proving properties of sorting algorithms, using a time limit of 5 minutes on machine with AMD Epyc 7502, 2.5 GHz CPU with 1 TB RAM, using 1 core and 16 GB RAM per benchmark.

Sortedness			
Benchm.	Pr.	T	Required lemmas
IS-S	✓	0.01	{IS-S-L1}
IS-S-L1	✓	8.28	-
MS-S	✓	0.08	∅
MS-S-L1	✓*	0	-
MS-S-L2	✓	0.02	∅
QS-S	✓	0.09	{QS-S-L1, QS-S-L2, QS-S-L3}, {QS-S-L1, QS-S-L3, QS-S-L4}
QS-S-L1	✓	0.27	∅
QS-S-L2	✓	0.04	{QS-S-L4}
QS-S-L3	✓	11.82	{QS-S-L4, QS-S-L5}
QS-S-L4	✓	8.28	{QS-S-L6}
QS-S-L5	✓	0	{QS-S-L7}
QS-S-L6	✓	0.02	∅
QS-S-L7	✓	0.02	∅

IS, MS and QS correspond to Insertionsort, Mergesort and Quicksort; S and PE respectively denote sortedness (3) and permutation equivalence (4), and Li stands for the i-th lemma of the problem.

runs of Vampire with structural/computation induction; these lemmas are then used as input assumptions to Vampire to prove validity of the respective benchmark.[2] A benchmark category SA-PR[-L$_i$] indicates that it belongs to proving the property PR for sorting algorithm SA, where PR is one of S (sortedness (3)) and PE (permutation equivalence (4)) and SA is one of IS (Insertionsort), MS (Mergesort) and QS (Quicksort). Additionally, an optional Li indicates that the benchmark corresponds to the i-th lemma for proving the property of the respective sorting algorithm.
For our experiments, we ran all possible combinations of lemmas to determine the minimal lemma dependency for each benchmark. For example, the sortedness property of Quicksort (QS-S) depends on seven lemmas (see Section 5.1), while the third lemma for this property (QS-S-L$_3$) depends on four lemmas (see Section 5.2). The second column Pr. indicates that Vampire solved the benchmark, by using a minimal subset of needed lemmas given in the fourth column. The third column T shows the running time in seconds of the respective saturation run using the first solving strategy identified during portfolio mode.
To identify the successful configuration, we ran Vampire in a portfolio setting for 5 minutes on each benchmark, with strategies enumerating all combinations of options that we hypothesized to be relevant for these problems. In accordance with Table 1, Vampire compositionally proves permutation equivalence of Insertionsort and Quicksort and sortedness of Mergesort and Quicksort. Note that sortedness of Mergesort is proven without any lemmas, hence lemma

[2] Link to experiments upon request due to anonymity.

$\texttt{MS-S-L}_1$ is not needed. The lemmas $\texttt{MS-PE-L}_1$ for the permutation equivalence of `Mergesort` and $\texttt{IS-S-L}_1$ for the sortedness of `Insertionsort` could be proven separately by more tailored search heuristics in VAMPIRE (hence ✓$*$), but our cluster setup failed to consistently prove these in the portfolio setting. Further statistics on inductive inferences are provided in Appendix A.

8 Related Work

While `Quicksort` has been proven correct on multiple occasions, not many have investigated a fully automated proof of the algorithm. One partially automated proof of `Quicksort`, closest to our work, relies on Dafny [15], where loop invariants are manually provided [2]. While [2] claims to prove some of these lemmas, not all invariants are proved correct (only assumed to be so). Similarly, the Why3 framework [4] has been leveraged to prove the correctness of `Mergesort` [16] over parameterized lists and arrays. These proofs also rely on manual proof splitting with the additional overhead of choosing the underlying prover for each subgoal as Why3 is interfaced with multiple automated and interactive solvers.
The work of [19] establishes the correctness of permutation equivalence for multiple sorting algorithms based on separation logic through inductive lemmas. However, [19] does not address the correctness proofs of the sortedness property. Contrarily, we automate the correctness proofs of sorting algorithms, using compositional first-order reasoning in the theory of parameterized lists.
Verifying functional correctness of sorting routines has also been explored in the abstract interpretation and model-checking communities, by investigating array-manipulating programs [6,9]. In [6], the authors automatically generate loop invariants for standard sorting algorithms of arrays of fixed length; the framework is, however, restricted solely to inner loops and does not handle recursive functions. Further, in [9] a priori given invariants/interpolants are used in the verification process. Unlike these techniques, we do not rely on a user-provided inductive invariant, nor are we restricted to inner loops.
There are naturally many examples of proofs of sorting algorithms using interactive theorem proving (ITP), see e.g. [10,13]. The work of [10] establishes correctness of `Insertionsort`. Similarly, the setting of [13] proves variations of `Introsort` and `Pdqsort` – both using Isabelle/HOL [20]. However, ITP relies on users to provide induction schemata, a burden that we eliminate in our approach.
When it comes to the landscape of automated reasoning, we are not aware of other techniques enabling fully automated verification of such sorting routines. To the best of our knowledge, the formal verification of `Quicksort` has so far not been automated, an open challenge which we solve in this paper.

9 Conclusion and Future Work

We present an integrated formal approach to establish program correctness over recursive programs based on saturation-based theorem proving. We automatically prove recursive sorting algorithms, particularly the `Quicksort` algorithm, by formalizing program semantics in the first-order theory of parameterized lists. Doing so, we expressed the common properties of sortedness and permutation equivalence in an efficient way for first-order theorem proving. By leveraging common structures of divide-and-conquer sorting algorithms, we advocate compositional first-order reasoning with built-in structural/computation induction.

We believe the implications of our work are twofold. First, integrating inductive reasoning in automated theorem proving to prove (sub)goals during interactive theorem proving can significantly alleviate the use of proof obligations to be shown manually, since automated theorem proving from our work can synthesize induction hypotheses to verify these conditions. Second, finding reasonable strategies to automatically split proof obligations on input problems can tremendously enhance the degree of automation in proofs that require heavy inductive reasoning. We hope that our work to open up future directions in combining interactive/automated reasoning, by further decreasing the amount of manual work in proof splitting, allowing thus superposition frameworks better applicable to a wider range of recursive algorithms. Proving further recursive sorting/search algorithms in future work, with improved compositionality, is therefore an interesting challenge to investigate.

Acknowledgements. This research was supported by the ERC Consolidator Grant ARTIST 101002685, the Austrian FWF SFB project SpyCoDe F8504, and the SecInt Doctoral College funded by TU Wien.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
2. Certezeanu, R., Drossopoulou, S., Egelund-Muller, B., Leino, K.R.M., Sivarajan, S., Wheelhouse, M.: Quicksort revisited: Verifying alternative versions of quicksort. Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday pp. 407–426 (2016)
3. Duarte, A., Korovin, K.: Ground joinability and connectedness in the superposition calculus. In: IJCAR. pp. 169–187. Springer (2022)
4. Filliâtre, J.C., Paskevich, A.: Why3–where programs meet provers. In: ESOP. pp. 125–128 (2013)
5. Foley, M., Hoare, C.A.R.: Proof of a recursive program: Quicksort. The Computer Journal **14**(4), 391–395 (1971)
6. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting Abstract Interpreters to Quantified Logical Domains. In: PoPL. pp. 235–246 (2008)
7. Hajdu, M., Hozzová, P., Kovács, L., Reger, G., Voronkov, A.: Getting saturated with induction. In: Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday, pp. 306–322. Springer (2022)
8. Hoare, C.A.: Quicksort. The computer journal **5**(1), 10–16 (1962)
9. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: CAV. pp. 193–206 (2007)

10. Jiang, D., Zhou, M.: A comparative study of insertion sorting algorithm verification. In: 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). pp. 321–325 (2017). https://doi.org/10.1109/ITNEC.2017.8284998
11. Kovács, L., Robillard, S., Voronkov, A.: Coming to Terms with Quantified Reasoning. In: POPL. pp. 260–270 (2017)
12. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: CAV. pp. 1–35 (2013)
13. Lammich, P.: Efficient verified implementation of introsort and pdqsort. In: IJCAR. pp. 307–323. Springer (2020)
14. Laneve, C., Montanari, U.: Axiomatizing permutation equivalence. Mathematical Structures in Computer Science **6**(3), 219–249 (1996)
15. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR. pp. 348–370 (2010)
16. Lévy, J.J.: Simple proofs of simple programs in why3. We thank all the contributors for their work, and Andrew Phillips for his editorial help. Martın Abadi Philippa Gardner p. 177 (2014)
17. Nipkow, T., Blanchette, J., Eberl, M., Gómez-Londoño, A., Lammich, P., Sternagel, C., Wimmer, S., Zhan, B.: Functional algorithms, verified (2021)
18. Robinson, A.J., Voronkov, A.: Handbook of automated reasoning, vol. 1. Elsevier (2001)
19. Safari, M., Huisman, M.: A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In: iFM. pp. 257–275. Springer (2020)
20. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: TPHOLs. pp. 33–38 (2008)

A Generated Inductive Inference during Proof Search

For all conjectures and lemmas that were proved in portfolio mode, we summarized the applications of inductive inferences with structural and computation induction schemata in Table 2. Specifically, Table 2 compares the number of inductive inferences performed during proof search (column `IndProofSearch`) with the number of used inductive inferences as part of each benchmark's proof (column `IndProof`). While most safety properties and lemmas required less than 50 inductive inferences, thereby using mostly one or two of them in the proof, some lemma proofs exceeded this by far. Most notably `IS-S-L1` and `QS-S-L1`, `Insertionsort`'s and `Quicksort`'s first lemma respectively, depended on many more inductive inferences until the right axiom was found. Such statistics point to areas where the prover still has room to be finetuned for software verification and quality assurance purposes, here especially towards establishing correctness of functional programs.