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`, $\texttt{filter}_<$ and $\texttt{filter}_\geq$ 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 $\texttt{filter}_<$ and $\texttt{filter}_\geq$ 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 inference 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`.

```
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.

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):

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

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 $sort :: a'list \to a'list$. 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:

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

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*:

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

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:

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

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:

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

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

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

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:

$$\forall xs_a .\, sorted(quicksort(xs)) \qquad (7)$$

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 $elem_{\leq}list :: a' \rightarrow a'list \rightarrow bool$ and $list_{\leq}list :: a'list \rightarrow a'list \rightarrow bool$:

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

and

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

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

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

• 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:

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

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:

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

(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:

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

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`:

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

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 $\mathit{elem}_{\leq}\mathit{list}(x, xs)$ and $\mathit{list}_{\leq}\mathit{list}(ys, \mathsf{cons}(x, xs))$ 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):

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

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 (`$\mathtt{a}_1$` ... `$\mathtt{a}_n$`) ...)` 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	$\emptyset$
`MS-PE`	✓	0.06	{`MS-PE-L1`, `MS-PE-L2`}
`MS-PE-L1`	✓*	0	-
`MS-PE-L2`	✓	0.03	$\emptyset$
`MS-PE-L3`	✓	0.15	$\emptyset$
`QS-PE`	✓	0.5	{`QS-PE-L1`, `QS-PE-L2`}
`QS-PE-L1`	✓	0.05	$\emptyset$
`QS-PE-L2`	✓	0.09	$\emptyset$

`Sortedness`			
Benchm.	Pr.	T	Required lemmas
`IS-S`	✓	0.01	{`IS-S-L1`}
`IS-S-L1`	✓	8.28	-
`MS-S`	✓	0.08	$\emptyset$
`MS-S-L1`	✓*	0	-
`MS-S-L2`	✓	0.02	$\emptyset$
`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	$\emptyset$
`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	$\emptyset$
`QS-S-L7`	✓	0.02	$\emptyset$

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. `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.

`MS-S-L`$_1$ is not needed. The lemmas `MS-PE-L`$_1$ for the permutation equivalence of `Mergesort` and `IS-S-L`$_1$ for the sortedness of `Insertionsort` could be proven separately by more tailored search heuristics in VAMPIRE (hence $\checkmark *$), 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.

Table 2: Structural Induction Applications in Proof Search and Proof.

Benchmark	IndProofSearch	IndProof
IS-S	4	1
IS-S-L1	339	2
IS-PE	5	1
IS-PE-L1	34	1
MS-S	8	1
MS-S-L2	22	1
MS-PE	14	1
MS-PE-L2	16	1
MS-PE-L3	136	3
QS-S	10	2
QS-S-L1	510	2
QS-S-L2	9	1
QS-S-L3	130	2
QS-S-L4	183	3
QS-S-L5	0	0
QS-S-L6	26	1
QS-S-L7	16	2
QS-PE	12	1
QS-PE-L1	10	1
QS-PE-L2	42	4