Sorting Without Sorts

Pamina Georgiou, Marton Hajdu and Laura Kovacs

July 29, 2023

Sorting without Sorts

Pamina Georgiou[1], Márton Hajdu[1], and Laura Kovács[1]

TU Wien, Austria

Abstract. We present an integrated formal methods framework in support of automatically establishing the functional correctness of programs with recursive data structures, including functional programs implementing sorting algorithms. We formalize program semantics in many-sorted first order logic while introducing sortedness/permutation properties as part of our first-order formalization. Rather than focusing on specific first-order theories such as lists of integer arithmetic, our formalization relies on a parameterized sort abstracting (arithmetic) theories. We further 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 a fully automated verification engine, without requiring manually proven and/or a priori given loop invariants.

1 Introduction

Sorting algorithms are ubiquitous in computing. They 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` [7] 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` would require additional recursive data structures such as stacks. While `Quicksort` and other sorting routines have been proven correct by means of manual efforts [4], proof assistants [15], abstract interpreters [5], or model checkers [8], to the best of our knowledge such correctness proofs so far have not been fully automated.

In this paper we aim to enforce the partial correctness of functional programs with recursive data structures, in a fully automated manner. The crux of our approach is a compositional reasoning setting based on superposition-based first-order theorem proving [11] with native support for induction [6] and first-order

```
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`, `filter`$_<$ and `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 `filter`$_<$ and `filter`$_\geq$ functions return lists of elements y of xs such that $y < x$ and $y \geq x$, respectively.

theories of recursively defined data types [10]. We extend this setting to support the first-order theory of list data structures parameterized by an abstract background theory/sort a. Doing so, we introduce a reasoning framework that integrates static program analysis, first-order theorem proving, and the automation of induction. 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 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 parametrized by a, allowing us to quantify over lists of sort a as they are domain elements of our first-order theory.

(ii) We leverage *first-order theorem proving for compositional proofs* of recursive parameterized sorting algorithms (Section 4), in particular of `Quicksort` from Figure 1. Our proofs do not rely on manually proven invariants. Rather, we embed the application of induction directly in saturation proving and we split (sorting) verification conditions using first-order lemmas, where each of these lemmas is proved with the help of saturation-based theorem proving. That is, all verification conditions of respective sorting algorithms 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 a priori given induction hypotheses and/or inductive invariants, but generate such inductive hypotheses/invariants as logical consequences (lemmas) of our program semantics.

(iii) We note that sorting algorithms often follow a divide-and-conquer approach (see Figure 2). We provide a *generalized set of lemmas* that is applicable to sorting algorithms on recursive data structures, such as lists (Section 5). 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. Permutation 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 formalize 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) Based on our first-order semantics of sorting algorithms, we showcase compositional reasoning via first-order theorem provers with built-in induction (Section 4). We exploit a divide-and-conquer approach implemented by sorting algorithms and provide a fully automated correctness proof of the recursive `Quicksort` algorithm of Figure 1.

(iii) We generalize our inductive lemmas to prove the functional correctness of multiple functional sorting algorithms (Section 5), including `Mergesort` and `Insertionsort`. We demonstrate our findings (Section 6) by implementing our approach on top of the Vampire theorem prover [11].

2 Preliminaries

We assume familiarity with standard first-order logic (FOL) and briefly introduce saturation-based proof search in first-order theorem proving [11].

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 $S = \mathcal{A} \cup \{\neg G\}$ is unsatisfiable[1]. To

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

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 $C \to D$. 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 $\mathcal{A} \to G$; 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 [16]. 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 [10]. 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 $\leq$; for simplicity, we also use $\geq$ and $<$ as the standard ordering versions of $\leq$. As a result, we work in the first-order theory of lists parametrized 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. 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.

Inductive Reasoning in Saturation. Inductive reasoning has recently been embedded in saturation-based theorem proving [6], by extending the superposition calculus with a new inference rule based on *induction axioms.* 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:

$$\big(F[\mathsf{nil}] \land \forall x, ys.(F[ys] \to F[\mathsf{cons}(x, ys)])\big) \to \forall zs.F[zs] \tag{1}$$

Sorting algorithms however often require induction axioms that are more complex than instances of structural induction (1). Such axioms are typically instances of *computation/recursion induction* schema, arising from sorting scenarios when, for example, the task of sorting a list is *reduced* to recursive applications of sorting tasks to two or more smaller lists (i.e. sublists). Divide-and-conquer sorting strategies mostly reduce lists into two complementary sublists based on some comparison operation $\circ$ among lists, using also the inverse $\circ^{-1}$ operation of $\circ$. We therefore use the following computation induction schema over lists:

$$\left(F[\mathsf{nil}] \land \forall x, ys. \left(\begin{pmatrix} F[reduce_{\circ}(x, ys)] \land \\ F[reduce_{\circ^{-1}}(x, ys)] \end{pmatrix} \to F[\mathsf{cons}(x, ys)]\right)\right) \to \forall zs.F[zs] \tag{2}$$

3 First-Order Semantics of Functional Sorting Algorithms

We outline our formalization of recursive sorting algorithms in the full first-order first-order theory of parametrized 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 `f`, as well as other recursive operations over lists implement a *divide-and-conquer* approach: (i) use a *reduction 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) call *combination function* that puts together the result of the recursive calls of `f`. Figure 2 shows such a divide-and-conquer pattern, where the reduction function *reduce* uses an invertible operator $\circ$, with $\circ^{-1}$ being the inverse 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(reduce$_\circ$(cons(y,ys))),
    f(reduce$_{\circ^{-1}}$(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 parametrized lists (in first-order logic, function definitions can be considered as universally quantified equalities):

$$\begin{aligned} f(\mathsf{nil}) &= \mathsf{nil} \\ \forall x_a, xs_a .\ f(\mathsf{cons}(x, xs)) &= combine(\ f(reduce_\circ(\mathsf{cons}(x, xs))), \\ &\qquad\qquad\quad f(reduce_{\circ^{-1}}(\mathsf{cons}(x, xs)))). \end{aligned} \tag{3}$$

The recursive divide-and-conquer pattern of Figure 2, together with the first-order semantics (3) of `f`, will be used in Sections 4-5 for respectively proving correctness of the `Quicksort` algorithm (and other sorting algorithms), as well as for applying lemma generalizations for divide-and-conquer list operations. The aware reader might however already notice that the computation induction schema (2) will be the necessary ingredient to automate inductive reasoning over `f` and their respective specifications. 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 behaviour 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:

$$\begin{gathered} sorted(\mathsf{nil}) = \top \\ \forall x_a, xs_a.\, sorted(\mathsf{cons}(x, xs)) = (elem_{\leq}list(x, xs) \land sorted(xs)), \end{gathered} \tag{4}$$

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:

$$\forall xs_a.\, sorted(sort(xs)). \tag{5}$$

(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 [13]. 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. 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 $\mathtt{filter}_=$ to the input/output list of *sort* are the same. Formally, we have the following first-order property of permutation equivalence:

$$\forall x_a, xs_a \,.\, filter_=(x, xs) = filter_=(x, sort(xs)). \tag{6}$$

4 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 therefore reduce the task of proving the functional correctness of `Quicksort` to the task of proving the (i) sortedness property (5) and (ii) the permutation equivalence property (6) of `Quicksort`. As mentioned in Section 3.2, a similar reasoning is needed for most sorting algorithms, which we evidence in Sections 5–6.

4.1 Proving Sortedness for `Quicksort`

Given an input list xs, we prove that `Quicksort` computes a sorted list by considering the property (5) instantiated for `Quicksort`. That is, we prove:

$$\forall xs_a \,.\, sorted(quicksort(xs)) \tag{7}$$

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 than, resp. greater-or-equal to an element x results in, smaller, resp. greater-or-equal sublists.

Similarly to (4), in order to express property **(S2)** we introduce the inductive predicates $elem_\leq list :: a' \to a'list \to bool$ and $list_\leq list :: a'list \to a'list \to bool$, defined respectively as:

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

to express that an element x is smaller than any list element of ys, and

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

to express that every element x in 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 (3) of `Quicksort`. Each of these lemmas is automatically proven by saturation-based theorem proving using the structural and/or computation induction schemata of (1) and (2); 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-or-equal than all elements in ys, the result of concatenating ys and xs yields a sorted list. Formally, we have

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

• 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, the list of $\texttt{filter}_{<}\texttt{(x,xs)}$ contains 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 $\mathsf{cons}(x, filter(x_{\geq}(x, xs)))$. By applying `Quicksort` over the input list xs, we thus have:

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

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`:

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

where axiom (13) is automatically obtained from the computation induction schema (2) by setting $F := sorted(quicksort(t))$ for some term t, $reduce_{\circ} := filter_{<}$ and $reduce_{\circ^{-1}} := filter_{\geq}$.

The first-order lemmas (10)–(12), together with the induction axiom (13) and the first-order semantics (3) of `Quicksort`, imply the sortedness property (6) 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 (1) 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,

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

and

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

• 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,

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

and

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

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 (3) 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 to be used during proof search.

4.2 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 (6) for `Quicksort`. That is, we prove:

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

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:

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

(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 inverse reduction operations. This property is expressed via lemma:

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

Similarly as in Section 4.1, we prove lemmas (19)–(20) by saturation-based reasoning with built-in induction. In particular, lemma (19) is established using the structural induction schema (1) in saturation, while the validity of lemma (20) is obtained by applying the computation induction schema (2) in saturation. By proving lemmas (19)–(20), we thus establish validity of permutation equivalence (18) for `Quicksort`. Together with the sortedness property (7) of `Quicksort` proven in Section 4.1, 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 to the first-order semantics (3) of `Quicksort`.

5 Lemma Generalization for Sorting

Establishing the functional correctness of `Quicksort` in Section 4 uses nine first-order lemmas that express inductive properties over lists in addition to the first-order semantics (3) 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. In this section, we further improve the automated proving support of Section 4: we show that the lemmas introduced in Section 4 can automatically be generated within saturation with induction. Moreover, we demonstrate that the lemmas of Section 4 can be leveraged to prove correctness of other divide-and-conquer list sorting algorithm, in particular within `Mergesort` (Figure 5). Finally, the genericity of our inductive lemmas from Section 4 helps also reasoning about sorting routines that do not follow a divide-and-conquer strategy, such as `Insertionsort` (Figure 4).

```
insertsort :: a’ list → a’ list
insertsort(nil) = nil
insertsort(cons(x, xs)) = isort(x, insertsort(xs))

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

```

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

5.1 Common Patterns of Inductive Lemmas over Sorting

Consider the computation induction schema (2). When using (2) for proving the sortedness (7) and permutation equivalence (18) of `Quicksort`, the inductive formula F of (2) is, respectively, instantiated with the predicates *sorted* from (7) and $filter_=$ from (18). The base case $F[\mathsf{nil}]$ of schema (2) is then trivially proved by saturation for both properties (7) and (18) of `Quicksort`.

Proving the induction step case of schema (2) is however challenging as it relies on *reduce*-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 (2) for the sortedness (7) and permutation equivalence (18) properties requires showing that applying *combine* functions over *reduce* functions preserve sortedness (7) and permutation equivalence (18), respectively. For divide-and-conquer algorithms of Figure 2, the step case of schema (2) requires thus proving the following lemma:

$$\left(\forall x_a, ys_a. \left(combine\begin{pmatrix} F[reduce_{\circ}(x, ys)]\land \\ F[reduce_{\circ^{-1}}(x, ys)]\end{pmatrix} \rightarrow F[\mathsf{cons}(x, ys)]\right)\right). \tag{21}$$

We next describe generic instances of lemma (21) to be used within proving functional correctness of sorting allgorithms, depending on the *combine*/*reduce* function of the underlining divide-and-conquer sorting routine.

(i) ***Combining sorted lists preserves sortedness.*** For proving the inductive step case (21) of the sortedness property (5) of sorting algorithms, we require the following generic lemma (5):

$$\forall xs_a, ys_a\textbf{.}\ \big(sorted(xs) \land sorted(ys)\big) \rightarrow sorted(combine(xs, ys)), \tag{22}$$

```
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 `reduce` 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).

ensuring that combining sorted lists results in a sorted list. Lemma (22) is used to establish property **(S1)** of `Quicksort`, namely used as lemma (10) for proving the preservation of sortedness under the *append* function.

Remark 1 (Compositional reasoning over sortedness in saturation). Note that applying lemma (22) directly to `Quicksort`, without taking into account the structure of the recursive calls of *append* and *filter*-functions, would results in the following weaker version of lemma (10):

$$\begin{aligned}&\forall x_a, xs_a, ys_a \,.\\ &\quad \big(sorted(xs) \wedge sorted(ys)\big) \rightarrow sorted(append(ys, \mathsf{cons}(x, xs)))\end{aligned} \tag{23}$$

which could automatically be derived by saturation with computation induction (2). However, lemma (23) is not valid 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). Therefore, the assumptions $elem_{\leq}list(x, xs)$ and $list_{\leq}list(ys, \mathsf{cons}(x, xs))$ are also needed in addition to (23), resulting in lemma (10). Yet, lemma (10) from Section 4 can automatically be derived via saturation with *compositional reasoning* based on computation induction (2): we derive (23) from (2) via saturation, strengthen (23) with $elem_{\leq}list(x, xs)$ and $list_{\leq}list(ys, \mathsf{cons}(x, xs))$ via superposition, yielding thus (10).

We showcase that genericity of lemma (22), by using it upon sorting routines different than `Quicksort`. Consider, for example, `Mergesort` as given in Figure 5. The sortedness property (5) of `Mergesort` can be proved by using saturation with lemma 22; note that the `merge` function of `Mergesort` acts as a *combine* function of (22). That is, we establish the sortedness property of `Mergesort` via the following instance of (22):

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

Finally, lemma (22) is not restricted to divide-and-conquer routines. For example, when proving the sortedness property (5) of the `Insertionsort` algorithm of Figure 4, we use saturation with lemma (22) applied to `isort`. As such, sortedness of `Insertionsort` is established by the following instance of (22):

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

(ii) ***Combining reductions preserves permutation equivalence.*** Similarly to Section 4.2, proving permutation equivalence (6) 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 $\mathit{filter}_=$:

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

Note that lemma (19) for `Quicksort` is an instance of (26), as the *append* function of `Quicksort` acts as a *combine* function of Figure 2.

- Similarly to **(P2)** for `Quicksort`, we ensure that, by combining (inverse) *reduction* functions, we preserve (6). That is,

$$\forall x_a, xs_a \,.\, \mathit{filter}_=(x, xs) = \mathit{combine}(\mathit{filter}_=(x, \mathit{reduce}_\circ(xs)), \mathit{filter}_=(x, \mathit{reduce}_{\circ^{-1}}(xs))) \tag{27}$$

Note that lemma (20) for `Quicksort` is an instance of (27), as the $\mathit{filter}_<$ and $\mathit{filter}_\geq$ functions correspond to the (inverse) *reduce* functions of Figure 2.

To prove the permutation equivalence (6) property of `Mergesort`, we use the functions `take` and `drop` as the *reduce* functions of lemmas (26)–(27). Doing so, we embed a natural number n argument (of sort $\mathbb{N}$) in lemmas (26)–(27), with n controling how many list elements are *taken* and *dropped*, respectively, in `Mergesort`. As such, the following instances of lemmas (26)–(27) are adjusted to `Mergesort`:

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

and

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

with lemmas (28)–(29) being proved via saturation. With these lemmas at hand, the permutation equivalence (6) of `Mergesort` is established, similarly to `Quicksort`.

Finally, the genericity of lemmas (26)–(27) naturally pays of when proving the permutation equivalence properrty (6) of `Insertionsort`. Here, we only use a simplified instance of (26) to prove (6) is preserved by the auxiliary function `isort`. That is, we use the following instance of (26):

$$\forall x_a, y_a, ys_a.\ \mathit{filter}_{=}(x, \mathsf{cons}(y, ys)) = \mathit{filter}_{=}(x, \mathit{isort}(y, ys)), \tag{30}$$

which is established by saturation with computation induction (2).

We conclude by emphasizing the generality of the lemmas (22) 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.

6 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 [11]. 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 `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*, $\mathit{filter}_{<}$ and $\mathit{filter}_{\geq}$, 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 [15], namely `Quicksort`, `Mergesort`, and `Insertionsort`. We show that the functional correctness of these sorting routines can be 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 (6), while `Sortedness` refers to the sortedness (5) property, together implying the functional correctness of the respective sorting algorithm. Here, the inductive lemmas of Sections 4–5 are proven in separate saturation runs

`PermEq`		
Benchm.	Pr.	Required lemmas
`IS-PE`	✓	{`IS-PE-L1`}
`IS-PE-L1`	✓	∅
`MS-PE`	✓	{`MS-PE-L1`, `MS-PE-L2`}
`MS-PE-L1`	✓*	-
`MS-PE-L2`	✓	∅
`MS-PE-L3`	✓	∅
`QS-PE`	✓	{`QS-PE-L1`, `QS-PE-L2`}
`QS-PE-L1`	✓	∅
`QS-PE-L2`	✓	∅

`Sortedness`		
Benchm.	Pr.	Required lemmas
`IS-S`	✓	{`IS-S-L1`}
`IS-S-L1`	✓*	-
`MS-S`	✓	∅
`MS-S-L1`	✓*	-
`MS-S-L2`	✓*	∅
`QS-S`	✓	{`QS-S-L1`, `QS-S-L2`, `QS-S-L3`}, {`QS-S-L1`, `QS-S-L3`, `QS-S-L4`}
`QS-S-L1`	✓	∅
`QS-S-L2`	✓	{`QS-S-L4`}
`QS-S-L3`	✓	{`QS-S-L4`, `QS-S-L5`}
`QS-S-L4`	✓	{`QS-S-L6`}
`QS-S-L5`	✓	{`QS-S-L7`}
`QS-S-L6`	✓	∅
`QS-S-L7`	✓	∅

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 (5) and permutation equivalence (6), and `Li` stands for the *i*-th lemma of the problem.

of Vampire with structural/computation induction; these lemmas are then used as input assumptions to Vampire to prove validity of the respective benchmark. 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 (5)) and `PE` (permutation equivalence (6)) 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 4.1), while the third lemma for this property (`QS-S-L`$_3$) depends on four lemmas (see Section 4.2). The second column `Pr.` indicates that Vampire solved the benchmark, by using a minimal subsets of needed lemmas given in the third column.

We ran Vampire on each benchmark in a portfolio setting for 5 minutes, with strategies enumerating all combinations of options that we hypothesized to be relevant for solving 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 `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 ✓∗), but our cluster setup failed to consistently prove these with the portfolio setting.

7 Related Work

While Quicksort has been proven correct on multiple occasions, first and foremost in the famous 1971's pen-on-paper proof by Foley and Hoare [4], not many have investigated a fully automated proof of the algorithm. The only partially automated proof of Quicksort, to the best of our knowledge, relies on Dafny [14], where loop invariants are manually provided [2]. While [2] claims to prove some of the lemmas/invariants, not all invariants are proved corrects (only assumed to be so). The work of [17] establishes the correctness of permutation equivalence for multiple sorting algorithms based on separation logic through inductive lemmas. However, [17] does not address the correctness proofs of the sortedness property. Contrarily, we fully automate the correctness proofs of sorting algorithms, using 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 [5, 8]. In [5], 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 [8] a priori given invariants/interpolants are used in the verification process. Unlike these techniques, we do not rely on a priori given 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. [9, 12]. The work of [9] establishes correctness of insertion sort. Similarly, the setting of [12] proves variations of Introsort and Pdqsort – both using Isabelle/HOL [18]. However, ITP relies on user guidance to provide induction schemes, a burden that we eliminate in our approach.

When it comes to the landscape of automated reasoning, we are not aware of our techniques enabling the fully automated verification of such sorting routines. Moreover, 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.

8 Conclusions

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. Proving further recursive sorting/search algorithms in future work, with improved compositionality, is an interesting line to investigate.

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. Foley, M., Hoare, C.A.R.: Proof of a recursive program: Quicksort. The Computer Journal **14**(4), 391–395 (1971)
5. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting Abstract Interpreters to Quantified Logical Domains. In: PoPL. pp. 235–246 (2008)
6. 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)
7. Hoare, C.A.: Quicksort. The computer journal **5**(1), 10–16 (1962)
8. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: CAV. pp. 193–206 (2007)
9. 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
10. Kovács, L., Robillard, S., Voronkov, A.: Coming to Terms with Quantified Reasoning. In: POPL. pp. 260–270 (2017)
11. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: CAV. pp. 1–35 (2013)
12. Lammich, P.: Efficient verified implementation of introsort and pdqsort. In: IJCAR. pp. 307–323. Springer (2020)
13. Laneve, C., Montanari, U.: Axiomatizing permutation equivalence. Mathematical Structures in Computer Science **6**(3), 219–249 (1996)
14. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR. pp. 348–370 (2010)
15. Nipkow, T., Blanchette, J., Eberl, M., Gómez-Londoño, A., Lammich, P., Sternagel, C., Wimmer, S., Zhan, B.: Functional algorithms, verified (2021)
16. Robinson, A.J., Voronkov, A.: Handbook of automated reasoning, vol. 1. Elsevier (2001)
17. 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)
18. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: TPHOLs. pp. 33–38 (2008)