Certification of Sorting Algorithms Using Theorema and Coq

Isabela Drămnesc, Tudor Jebelean, Sorin Stratulat

Certification of Sorting Algorithms Using *Theorema* and Coq

Isabela Drămnesc[1], Tudor Jebelean[2], and Sorin Stratulat[3]

[1] Department of Computer Science, West University of Timisoara, Romania
Isabela.Dramnesc@e-uvt.ro
[2] ICAM, West University of Timisoara, Romania
RISC, Johannes Kepler University, Linz, Austria
Tudor.Jebelean@e-uvt.ro
[3] Université de Lorraine, CNRS, LORIA, Metz, F-57000, France
sorin.stratulat@univ-lorraine.fr

Abstract. Sorting is an operation that has very important practical applications, in particular for instance in the storage and analysis of data related to the environment, climat change, etc.
We describe experiments of automated certification of various sorting algorithms by using the *Theorema* as well as Coq and we compare the two techniques. The sorting algorithms are: *Quick–Sort*, *Patience–Sort*, *Min–Sort*, *Max–Sort*, and *Min–Max–Sort*. In both systems we construct the appropriate underlying theory, we define the algorithms in functional style, we run them on examples and we produce the proof of their correctness together with proofs of various lemmas that are necessary.
In *Theorema* the proofs are almost completely automatic, are presented in natural style, and the underlying theory uses multisets in order to express the fact that the input and the output have the same elements. Moreover the proofs use a generalized induction scheme based on the well-founded ordering on lists defined by strict inclusion of multisets.
In Coq the proofs are based on scripts, they require more additional lemmas, and they use element counts to compare the contents of the list. However, both the algorithm definitions as well as the proofs are absolutely rigurous as Coq cannot accept any elements that are not theoretically correct.

1 Introduction

Sorting algorithms play a fundamental role in various computational tasks, serving as the backbone for a wide range of applications across diverse domains, in particular in the ones related to environment, climate change, etc, due to the huge size of the data that has to be indexed and processed. As the volume and complexity of data continue to grow exponentially, the efficiency and correctness of sorting algorithms becomes imperative due to various reasons such as: data integrity and accuracy, performance optimization, security, and algorithmic transparency.

We focus on the automation of the verification process of sorting algorithms and of the auxiliary sub-algorithms. We develop proof–based techniques in the *Theorema* system for algorithm certification, we use the Coq interactive theorem prover for the verification of the same algorithms, and finally we compare the two approaches.

The *Theorema* system [8,9,37] is a complex framework implemented in *Mathematica*[4]. The system allows to define mathematical theories, including definition of algorithms by logical formulae, to experiment by running the algorithms, and to develop and use automatic provers. The system facilitates the verification of algorithms because their implementation in *Theorema* does not use a programming language, but they are defined directly in predicate logic together with their specification, thus there is no need to generate verification conditions. A main feature of the *Theorema* system is to generate proofs in natural style: both the inference rules and the presentation of the proofs imitate the human style.

The Coq system [35,4] is a state-of-the-art proof assistant that can be used to formalize algorithms and, the same time, to define and prove properties about them, by using a higher-order language called Calculus of Inductive Constructions. Based on the Curry-Howard isomorphism [23], the properties to be proved are *types* and the proofs are *terms* having as types the properties they prove. A crucial specificity of Coq is its *kernel* that implements an algorithm which checks whether a proof term has as type the property it proves. Thanks to this type-checking algorithm, the Coq proofs are reputed to be correct and Coq is widely used to certify algorithms.

Related Work. Formal certification environments like Coq [4] and Isabelle/HOL [28] have been used to certify classical algorithms on arrays/lists [19,36,29,38,10,5,30,34]. Other approaches use Krakatoa [27] in [36], SPARK* [22] in [2], PVS [33] in [12], KeY [1] in [6], and ACL2 [25] in [32].

Concerning the differences between the various approaches, the definitions of the algorithms and the definitions of sortedness are basically syntactic versions of the same functions and predicates expressed in functional style, while some other presentations [3,31,21] use imperative constructs. However the definition of the concept "have the same elements" (the input list vs. the sorted list) has several realizations: [7] uses a function that deletes the first occurrence of an element, [24] uses indices in arrays and the existence of a bijection, [11] counts elements, [20] compares the lists obtained by filtering individual elements, and finally only [10] uses multisets. These differences have a significant impact on the corresponding proofs, and these are typically more complex than the proofs related to sortedness.

Furthermore the proofs differ in the choice of the inference rules and proof strategies, as well as in the number of the necessary auxiliary notions and lemmas, which influences the length of the full proving process, its degree of automation, naturalness, understandability, etc. Therefore the research in this area is

[4] www.wolfram.com/mathematica

continuing to improve various aspects of mechanical verification, and the present paper aims to contribute to this trend.

Natural style (namely the approach in which the formulae, the inference steps, and the proof text are similar to the ones used by humans) is addressed in relatively few works: ours on verification and synthesis [13,15], as well as [7]. Also, there is no previous work in Coq on the verification of the algorithms presented here, see e.g. [19]. Similar work has been done by the authors on sorting algorithms for binary trees [14,18,16].

The novelty of this paper consists in:

- computer based certification of some important sorting algorithms;
- systematic comparison of the certification process in two systems that are quite different in their approach;
- development in *Theorema* of proof techniques that allow to produce relatively short proofs that are also in natural style;
- the use in *Theorema* of multisets and of inference rules that are relevant to them and to lists;
- generation and certification in Coq of additional properties about the sorting algorithms and auxiliary functions.

2 Context and Notations

The correctness of a sorting algorithm consists of two conditions on the output list: (a) it is sorted and (b) has the same elements as the input list. In *Theorema*, (b) is expressed using multisets, and in Coq using the number of occurrences.

Notations in *Theorema*. We consider multisets and lists over a totally ordered domain. We use uppercase roman letters like U, V, T for lists. A list is either empty $\langle\rangle$, or of the form head–tail denoted by $a \frown U$, (a is the head and U is the tail of the list – which is always a list). The elements of a list or a multiset are objects from a totally ordered domain (notation $<$ and $\leq$), and are denoted by lowercase roman letters like a, b, c, x. $\frown$ adds an element at the end of a list ($U \frown a$). For multisets we use the notations: $\emptyset$ for the empty set, $\{\{a\}\}$ for the multiset containing the element a occurring once, and $\mathcal{M}[U]$ for the multiset of the list U. The additive union of multisets is denoted by $\uplus$ (keeps the multiplicity of elements), like in [26]. The total ordering is extended to *element–list* (*element* is smaller than each member of *list*), *list–element* (each member of *list* is smaller than *element*), and $list_1$–$list_2$ (each member of $list_1$ is smaller than each element of $list_2$).

The type of the objects is not used explicitly, but it is automatically detected by the prover according to the notation conventions mentioned above.

In *Theorema*, function and predicate applications use squared brackets (e.g., $F[x], P[x]$). The quantified variables are written under the quantifier (e.g. $\underset{X}{\forall}$ "for all X", $\underset{X}{\exists}$ "exists X"), and Skolem constants have integer indices (e.g., U_0, a_0).

Notations in Coq. We represent the multisets of naturals as lists, typed **list nat**, and the In and count functions for defining the permutation relation between them. The constructors for **list** (resp., **nat**) are nil and :: (the notation for cons) (resp., 0 and S (the successor function)).

The recursive functions, as count, should be total and terminating. They are based on the `match` construct that defines different cases according to the constructors of the type of the function argument given as matching expression. Coq automatically checks the totality property by analysing the `match` cases and how they cover all the possible values the matching expression can take. Checking the termination property may be more intricate, requiring that some function argument should decrease after each recursive call w.r.t. some well-founded order. The `Fixpoint` keyword is used for defining the recursive functions for which Coq automatically identifies the recursive function argument and the well-founded order, otherwise the keyword `Function` is used. For the last case, the argument for recursion and the well-founded order are explicitly defined using the `wf` keyword, and the user has to provide the proofs for the termination conditions. The predicates are defined inductively, using the `Inductive` keyword and in terms of *axioms* that play the role of *constructors*.

3 The Algorithms

We certify the following sorting algorithms (that were synthesized in [15]):

1. *Quick–Sort* with auxiliary functions: *Concat[U,V]* (U concatenated with V), *LoE[U,a]* (the elements of U that are smaller than or equal to a), *High[U,a]* (the elements of U that are strictly greater than a);
2. *Patience–Sort* with auxiliary: *SelSort[U]*, *SelRest[U]* (split U in two lists: one list that is already sorted, and the other list containing the remaining elements), and *Merge[U,V]* (merge two sorted lists into a sorted one);
3. *Min-Sort*, with auxiliary: *min[U]* (the minimum of U), and *Trimmin[U]* (remove the minimum from U);
4. *Max-Sort*, with auxiliary: *max[U]* and *Trimmax[U]* – uses maximum.
5. *Min-Max-Sort*, auxiliary: *min[U]*, *max[U]*, and *TrimMM[U]* (remove the minimum and the maximum of U).

We present now the definition of the algorithm *Min-Max-Sort* and its auxiliary functions in the *Theorema* system.

Definition 3.1. *Min-Max-Sort.*

$$\underset{a,b,U}{\forall}\left(\begin{array}{c} MMS[\langle\rangle] = \langle\rangle \\ MMS[a \frown \langle\rangle] = a \frown \langle\rangle \\ MMS[a \frown (b \frown U)] = min[a \frown (b \frown U)] \frown (MMS[\,TrimMM[a \frown (b \frown U)]]) \\ \frown\, max[a \frown (b \frown U)] \end{array}\right)$$

Note that in the *Theorema* system one abbreviates several universal quantifiers by a single one containing several variables, and one can also indicate some simple condition on these variables in a second line under the quantifier. Furthermore

such a quantifier can prefix a conjunction of several formulas that are listed vertically without the conjunction symbol.

The above definition of *Min-Max-Sort* returns the sorted version of the input list. It places the minimum of the input list at the beginning of the output and the maximum at the end of the it, and then applies recursively to the list of the remaining elements, selected by the function *TrimMM* shown below.

Definition 3.2. *TrimMM.*

$$\left(\begin{array}{l} \underset{\substack{a,b,U \\ a\leq b}}{\forall}(TrimMM[a \frown (b \frown U)] = TrimMmA[a,b,U]) \\ \underset{\substack{a,b,U \\ b<a}}{\forall}(TrimMM[a \frown (b \frown U)] = TrimMmA[b,a,U]) \end{array}\right)$$

TrimMM calls the auxiliary function *TrimMmA* that returns the list without the minimum and the maximum elements from the input list.

Definition 3.3. *TrimMmA.*

$$\left(\begin{array}{l} \underset{a,b}{\forall}(TrimMmA[a,b,\langle\rangle] = \langle\rangle) \\ \underset{\substack{a,b,c,U \\ c<a}}{\forall}(TrimMmA[a,b,c \frown U] = a \frown TrimMmA[c,b,U]) \\ \underset{\substack{a,b,c,U \\ (a\leq c \wedge c\leq b)}}{\forall}(TrimMmA[a,b,c \frown U] = c \frown TrimMmA[a,b,U]) \\ \underset{\substack{a,b,c,U \\ b<c}}{\forall}(TrimMmA[a,b,c \frown U] = b \frown TrimMmA[a,c,U]) \end{array}\right)$$

Definition 3.4. *min.*

$$\left(\underset{a,U}{\forall}(min[a \frown U] = minA[a,U])\right)$$

The above function calls the following tail recursive function that gives the minimum element from a list.

Definition 3.5. *minA.*

$$\left(\begin{array}{l} \underset{a}{\forall}(minA[a,\langle\rangle] = a) \\ \underset{\substack{a,b,U \\ a\leq b}}{\forall}(minA[a,b \frown U] = minA[a,U]) \\ \underset{\substack{a,b,U \\ b<a}}{\forall}(minA[a,b \frown U] = minA[b,U]) \end{array}\right)$$

Definition 3.6. *max.*

$$\left(\underset{a,U}{\forall}(max[a \frown U] = maxA[a,U])\right)$$

The above function calls the following tail recursive function that gives the maximum element from a list.

Definition 3.7. *maxA.*

$$\begin{pmatrix} \forall\limits_{a}(maxA[a,\langle\rangle] = a) \\ \forall\limits_{\substack{a,b,U \\ a\leq b}}(maxA[a, b \frown U] = maxA[b, U]) \\ \forall\limits_{\substack{a,b,U \\ b<a}}(maxA[a, b \frown U] = maxA[a, U]) \end{pmatrix}$$

4 Certification in *Theorema*

4.1 Basic definitions.

The predicate *IsSorted* defined below returns true iff the input list is sorted.

Definition 4.1.

$$\forall\limits_{a,U}\begin{pmatrix} IsSorted[\langle\rangle] \\ IsSorted[a \frown U] \Longleftrightarrow (a \leq U \wedge IsSorted[U]) \end{pmatrix}$$

The function $\mathcal{M}$ defined below retuns the multiset of a list.

Definition 4.2. $\forall\limits_{a,U}\begin{pmatrix} \mathcal{M}[\langle\rangle] = \emptyset \\ \mathcal{M}[a \frown U] = \{\{a\}\} \uplus \mathcal{M}[U] \end{pmatrix}$

4.2 Special Inference Rules and Strategies.

The *Theorema* proofs use general inference rules from basic logic, but also certain special inference rules specific to the domain of lists and multisets.

The general rules are classical inference rules similar to those of sequent calculus, and other straightforward natural rules like applying properties of equality, using definitions, replacement by equal terms, etc. We list below the more important general rules, which are specific to our prover.

G1: *Generalized induction.* When proving the goal $\varphi[X]$, we may assume $\varphi[Y]$ for any Y strictly smaller than X with respect to strict inclusion of multisets ($X \subset Y$).

G2: *Cascading.* When the proof fails, construct a conjecture from the current proof state and prove it separately. This step requires the intervention of a human because it is not completely automated.

G3: *Contracting.* Reduce repeating composite constant term to constant.

G4: *Equality reduction.* Delete identical terms that occur on both sides of an equality goal.

Below are the most important inference rules that are specific to the domains of lists and multisets.

S1: *List expansion.* When it is assumed that a list X is nonempty, it is replaced by $a \frown Y$, where a and Y are new Skolem constants, and similarly for lists containing at least two elements.

S2: *Multiset expansion.* The term $\mathcal{M}[T]$ where T is a composite term representing a list can be expanded into several $\mathcal{M}$ terms by using multiset union. This is used mostly in conjunction with the rule **G4**.

S3: *Inequality reduction.* When the goal is an inequality (with respect to the order used for sorting) between lists, a term T representing a list whose multiset is known to be included in the multiset of another list X can be replaced by X.

Some of these rules have been used in [17] for verification of *Merge-Sort* and *Insert–Sort*, namely **G1**, **G2** and **G4**.

4.3 The Certification of *Min-Max-Sort*.

For the certification of *Min-Max-Sort* algorithm we have to prove the two properties: that the algorithm preserves multisets and that the output is sorted. The proofs are summarized in the next subsections.

***Min-Max-Sort* preserves multisets.**

Conjecture 4.3. $\underset{X}{\forall}\Big(\mathcal{M}[X] = \mathcal{M}[MMS[X]]\Big)$

For proving this conjecture, it is necessary to use the following two properties (which were generated by cascading and were proved – the proofs are detailed later in the paper):

Proposition 4.4. $\underset{U\neq\langle\rangle}{\forall}(\mathcal{M}[U] = \{\{min[U]\}\} \uplus \mathcal{M}[TrimMM[U]] \uplus \{\{max[U]\}\})$

Proposition 4.5. $\underset{U\neq\langle\rangle}{\forall}(\mathcal{M}[TrimMM[U]] \subset \mathcal{M}[U])$

Proof. Prove Conjecture 4.3 by cases using Definition 3.1.
Case 1. $X = \langle\rangle, MMS[\langle\rangle] = \langle\rangle$ trivial.
Case 2. $X = a_0 \frown \langle\rangle, MMS[a_0 \frown \langle\rangle] = a_0 \frown \langle\rangle$ trivial.
Case 3. $X = a_0 \frown (b_0 \frown U_0), MMS[a_0 \frown (b_0 \frown U_0)] = min[a_0 \frown (b_0 \frown U_0)] \frown MMS[TrimMM[a_0 \frown (b_0 \frown U_0)]] \frown max[a_0 \frown (b_0 \frown U_0)]$

Prove

$$\begin{aligned}\mathcal{M}[a_0 \frown (b_0 \frown U_0)] = \mathcal{M}[min[a_0 \frown (b_0 \frown U_0)] \frown \\ MMS[TrimMM[a_0 \frown (b_0 \frown U_0)]] \frown max[a_0 \frown (b_0 \frown U_0)]].\end{aligned} \tag{1}$$

By Definition 4.2 the goal becomes:

$$\begin{aligned}\{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \mathcal{M}[U_0] = \{\{min[a_0 \frown (b_0 \frown U_0)]\}\} \uplus \\ \mathcal{M}[MMS[TrimMM[a_0 \frown (b_0 \frown U_0)]]] \uplus \{\{max[a_0 \frown (b_0 \frown U_0)]\}\}\end{aligned} \tag{2}$$

By Proposition 4.5 and by **G1** the goal becomes:

$$\begin{aligned}\{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \mathcal{M}[U_0] = \{\{min[a_0 \frown (b_0 \frown U_0)]\}\} \uplus \\ \mathcal{M}[TrimMM[a_0 \frown (b_0 \frown U_0)]] \uplus \{\{max[a_0 \frown (b_0 \frown U_0)]\}\}\end{aligned} \tag{3}$$

By Proposition 4.4 the goal is proved.

The proof of Proposition 4.4 is done by cases using Definition 3.2.

Proof. Case 1: $U_0 = a_0 \backsim (b_0 \backsim \langle\rangle)$ (the list has exactly two elements) is straightforward without induction, because in this case *TrimMM* reduces to the empty set.

Case 2: $U_0 = a_0 \backsim (b_0 \backsim (c_0 \backsim V_0))$ (the list has more than two elements)

Using Definitions 3.4, Definition 3.6, and Definition 3.3 the following goal is proved by cases

$$\begin{array}{c}\{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \{\{c_0\}\} \uplus \mathcal{M}[V_0] = \{\{min[a_0 \backsim (b_0 \backsim (c_0 \backsim V_0))]\}\} \uplus \\ \mathcal{M}[TrimMM[a_0 \backsim (b_0 \backsim (c_0 \backsim V_0))]] \uplus \{\{max[a_0 \backsim (b_0 \backsim (c_0 \backsim V_0))]\}\}\end{array} \tag{4}$$

Case 2.1.: $a_0 \leq b_0$

The goal (4) is

$$\begin{array}{c}\{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \{\{c_0\}\} \uplus \mathcal{M}[V_0] = \{\{minA[a_0, c_0 \backsim V_0]\}\} \uplus \\ \mathcal{M}[TrimMmA[a_0, b_0, c_0, V_0]] \uplus \{\{max[b_0, c_0 \backsim V_0]\}\}\end{array} \tag{5}$$

Using Definition 3.3 we prove by cases: *Case 2.1.1.* $V_0 = \langle\rangle$
Case 2.1.1.1 $c_0 < a_0$ the goal is:

$$\{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \{\{c_0\}\} \uplus \emptyset = \{\{c_0\}\} \uplus \mathcal{M}[a_0 \backsim TrimMmA[c_0, b_0, \langle\rangle]] \uplus \{\{b_0\}\} \tag{6}$$

which is

$$\{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \{\{c_0\}\} \uplus \emptyset = \{\{c_0\}\} \uplus \{\{a_0\}\} \uplus \emptyset \uplus \{\{b_0\}\} \tag{7}$$

This is true by commutativity of multiset union.

For the following two cases, namely, *Case 2.1.1.2* $a_0 \leq c_0 \wedge c_0 \leq b_0$ and *Case 2.1.1.3* $b_0 < c_0$ the proofs are similar to *Case 2.1.1.1.*

Case 2.1.2. $c_0 < a_0$ From the assumptions, by transitivity, we know

$$c_0 < b_0 \tag{8}$$

The goal (5) becomes:

$$\begin{array}{c}\{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \{\{c_0\}\} \uplus \mathcal{M}[V_0] = \{\{minA[c_0, V_0]\}\} \uplus \\ \{\{a_0\}\} \uplus \mathcal{M}[TrimMmA[b_0, c_0, V_0]] \uplus \{\{maxA[b_0, V_0]\}\}\end{array} \tag{9}$$

By **G4** the goal becomes:

$$\begin{array}{c}\{\{b_0\}\} \uplus \{\{c_0\}\} \uplus \mathcal{M}[V_0] = \{\{minA[c_0, V_0]\}\} \uplus \\ \mathcal{M}[TrimMmA[b_0, c_0, V_0]] \uplus \{\{maxA[b_0, V_0]\}\}\end{array} \tag{10}$$

This is true by **G1** because $b_0 \backsim (c_0 \backsim V_0)$ is strictly included in $a_0 \backsim (b_0 \backsim (c_0 \backsim V_0))$.

For the following two cases, namely *Case 2.1.3.* $a_0 \leq c_0 \wedge c_0 \leq b_0$ and *Case 2.1.4.* $b_0 < c_0$, the proofs are similar to *Case 2.1.2.*.

Case 2.2.: $b_0 < a_0$ As in *Case 2.1* four main cases are generated, namely *Case 2.2.1.* $V_0 = \langle\rangle$, *Case 2.2.2.* $c_0 < b_0$, *Case 2.2.3.* $b_0 \leq c_0 \wedge c_0 \leq a_0$ and *Case 2.2.4.* $a_0 < c_0$. The proofs in these cases are similar to the previous cases shown.

The proof of Proposition 4.5 is done by cases using Definition 3.2.

Proof. Using **S1**, the goal is

$$\mathcal{M}[\mathit{TrimMM}[a_0 \frown (b_0 \frown V_0)]] \subset \mathcal{M}[a_0 \frown (b_0 \frown V_0)]) \tag{11}$$

By Definition 3.2 we have two cases:

Case 1: $a_0 \leq b_0$, $\mathit{TrimMM}[a_0 \frown (b_0 \frown V_0)] = \mathit{TrimMmA}[a_0, b_0, V_0]$. And four more cases are generated using Definition 3.3, namely *Case 1.1.* $V_0 = \langle\rangle$, *Case 1.2.* $V_0 = c_0 \frown W_0, c_0 < a_0$, *Case 1.3.* $V_0 = c_0 \frown W_0, a_0 \leq c_0 \wedge c_0 \leq b_0$ and *Case 1.4.* $V_0 = c_0 \frown W_0, b_0 < c_0$. The proofs are similar in all cases. We summarize for *Case 1.4.*. By transitivity we know $a_0 < c_0$. By Definition 4.2, Definition 3.3 the goal becomes:

$$\{\{b_0\}\} \uplus \mathcal{M}[\mathit{TrimMmA}[a_0, c_0, W_0]] \subset \{\{a_0\}\} \uplus \{\{b_0\}\} \uplus \{\{c_0\}\} \uplus \mathcal{M}[W_0]) \tag{12}$$

By **G4** the goal becomes:

$$\mathcal{M}[\mathit{TrimMmA}[a_0, c_0, W_0]] \subset \{\{a_0\}\} \uplus \{\{c_0\}\} \uplus \mathcal{M}[W_0]) \tag{13}$$

which is true by **G1** because $a_0 \frown (b_0 \frown W_0)$ is strictly included in $a_0 \frown (b_0 \frown (c_0 \frown W_0))$.

Case 2: $b_0 < a_0$, $\mathit{TrimMM}[a_0 \frown (b_0 \frown V_0)] = \mathit{TrimMmA}[b_0, a_0, V_0]$. Similarly, four more cases are generated using Definition 3.3, namely *Case 2.1.* $V_0 = \langle\rangle$, *Case 2.2.* $V_0 = c_0 \frown W_0, c_0 < b_0$, *Case 2.3.* $V_0 = c_0 \frown W_0, b_0 \leq c_0 \wedge c_0 \leq a_0$ and *Case 2.4.* $V_0 = c_0 \frown W_0, a_0 < c_0$. The proofs for all these cases are similar to the ones shown before.

***Min-Max-Sort* sorted output.**

Conjecture 4.6. $\underset{X}{\forall}\Big(\mathit{IsSorted}[\mathit{MMS}[X]]\Big)$

For proving this conjecture it is necessary to use the following properties:

Proposition 4.7. $\underset{X}{\forall}(\mathit{min}[X] \leq X)$

Proposition 4.8. $\underset{X}{\forall}(X \leq \mathit{max}[X])$

Proposition 4.9. $\underset{X}{\forall}(\mathit{min}[X] \leq \mathit{max}[X])$

Proposition 4.10. $\underset{a,X}{\forall}(\mathit{IsSorted}[X \frown a] \iff \mathit{IsSorted}[X] \wedge X \leq a)$

Proposition 4.11.
$\underset{X}{\forall}((\mathit{min}[X] \leq X \wedge \mathit{min}[X] \leq \mathit{max}[X]) \implies \mathit{min}[X] \leq X \frown \mathit{max}[X]))$

Proof. Case 1. $X_0 = \langle\rangle$ trivial. *Case 2.* $X_0 = a_0 \backsim \langle\rangle$ trivial.
Case 3. $X_0 = a_0 \backsim (b_0 \backsim U_0)$ The goal is:

$$\mathit{IsSorted}[\mathit{min}[X_0] \backsim \mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0]] \tag{14}$$

By Definition 4.1 the goal reduces to:

$$\begin{aligned} \mathit{min}[X_0] \leq \mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0] \wedge \\ \mathit{IsSorted}[\mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0]] \end{aligned} \tag{15}$$

By Proposition 4.11 the goal reduces to

$$\begin{aligned} \mathit{min}[X_0] \leq \mathit{MMS}[\,\mathit{TrimMM}[X_0]] \wedge \mathit{min}[X_0] \leq \mathit{max}[X_0] \wedge \\ \mathit{IsSorted}[\mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0]] \end{aligned} \tag{16}$$

By Proposition 4.4 (*TrimMM* preserves multisets) and **S3** the goal reduces to:

$$\begin{aligned} \mathit{min}[X_0] \leq \mathit{TrimMM}[X_0] \wedge \mathit{min}[X_0] \leq \mathit{max}[X_0] \wedge \\ \mathit{IsSorted}[\mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0]] \end{aligned} \tag{17}$$

By Proposition 4.5 and **S3** the goal reduces to:

$$\begin{aligned} \mathit{min}[X_0] \leq X_0 \wedge \mathit{min}[X_0] \leq \mathit{max}[X_0] \wedge \\ \mathit{IsSorted}[\mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0]] \end{aligned} \tag{18}$$

By Proposition 4.7 the goal reduces to:

$$\mathit{min}[X_0] \leq \mathit{max}[X_0] \wedge \mathit{IsSorted}[\mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0]] \tag{19}$$

By Proposition 4.9 the goal reduces to:

$$\mathit{IsSorted}[\mathit{MMS}[\,\mathit{TrimMM}[X_0]] \frown \mathit{max}[X_0]] \tag{20}$$

By Proposition 4.10 the goal reduces to:

$$\mathit{IsSorted}[\mathit{MMS}[\,\mathit{TrimMM}[X_0]]] \wedge \mathit{MMS}[\,\mathit{TrimMM}[X_0]] \leq \mathit{max}[X_0] \tag{21}$$

By Proposition 4.5 and **G1** the goal reduces to:

$$\mathit{MMS}[\,\mathit{TrimMM}[X_0]] \leq \mathit{max}[X_0] \tag{22}$$

By Conjecture 4.3 (*Min-Max-Sort* preserves multisets) and **S3** the goal reduces to:

$$\mathit{TrimMM}[X_0] \leq \mathit{max}[X_0] \tag{23}$$

By Proposition 4.5 and and **S3** the goal reduces to:

$$X_0 \leq \mathit{max}[X_0] \tag{24}$$

This holds by Proposition 4.8.

For space reasons we cannot describe in detail the proofs of the other sorting algorithms (*Quick–Sort, Patience–Sort, Min–Sort*, and *Max–Sort*), but in fact the proof presented here is the most complex. The other proofs have a similar structure: two main theorems that ensure sortedness and the permutation property, and the lemmas that express the corresponding properties of the auxiliary algorithms.

5 Certification in Coq

We have used Coq to certify the algorithms from Section 3. Generally speaking, for each algorithms, we proved that they are *sorting* algorithms, i.e., the output list is a permutation of the input list that is also sorted (in this paper, in increasing order). Due to the lack of space, we will detail the certification process only for *Min-Max-Sort*.

The output list is checked to be sorted by the **sorted** inductive predicate:

```
Inductive sorted : list nat → Prop :=
  snil : sorted nil
| s1 : ∀ x, sorted (x::nil)
| s2 : ∀ x y l, sorted (y::l) → x ≤ y → sorted (x::y::l).
```

The permutation relation is defined using the recursive function count, for counting the number of occurrences of an element in a list, the boolean equality =? and the builtin membership predicate In:

```
Fixpoint count x l :=
  match l with
    nil ⇒ 0
  | hd :: tl ⇒ if x =? hd then S (count x tl) else count x tl
  end.

Definition permutation l l' := ∀ x, (In x l ↔ In x l') ∧ count x l = count x l'.

Definition is_a_sorting_algorithm (f: list nat → list nat) :=
  ∀ l, permutation (f l) l ∧ sorted (f l).
```

The definition of the *Min-Max-Sort* algorithm, as formalized by the MMS function displayed below, is based on the recursive functions maxA, minA and TrimMMA and the builtin functions Nat.lt, Nat.ltb, Nat.leb, length, app, Nat.min and Nat.max standing for, respectively, $<$ on naturals (**nat**) and its boolean version, the boolean $\leq$, the length of a list (**list**), the concatenation of two lists, the minimum and the maximum of two naturals:

```
Fixpoint maxA x l :=
  match l with
    nil ⇒ x
  | hd :: tl ⇒
      if Nat.leb x hd
      then maxA hd tl
      else maxA x tl
end.
```

```
Fixpoint minA x l :=
  match l with
    nil ⇒ x
  | hd :: tl ⇒
      if Nat.leb x hd
      then minA x tl
      else minA hd tl
end.
```

```
Fixpoint TrimMMA a b l :=
  match l with
    nil ⇒ nil
  | c :: tl ⇒
      if Nat.ltb c a
      then a :: (TrimMMA c b tl)
      else
        if Nat.leb c b
        then c :: (TrimMMA a b tl)
        else b :: (TrimMMA a c tl)
  end.
```

```
Function MMS l {
  wf (fun (l1: list nat) (l2: list nat) ⇒
Nat.lt (length l1) (length l2)) l} :=
  match l with
    nil ⇒ nil
  | hd :: tl ⇒
      (match tl with
         nil ⇒ hd :: nil
       | hd' :: tl' ⇒
           app ((minA hd tl) ::
  (MMS (TrimMMA (Nat.min hd hd')
            (Nat.max hd hd') tl')))
       (MMS ((maxA hd tl) :: nil))
       end)
  end.
```

As a proof example in Coq, we give the complete proof script for the lemma sorted_sorted which states that if a non-empty list is sorted then the tail of the list is also sorted:

```
Lemma sorted_sorted: ∀ a l, sorted (a::l) → sorted l.
Proof.
  induction l; simpl; intros.
  - apply snil.
  - inversion H. trivial.
Qed.
```

The proof of sorted_sorted starts by generating the *proof state* which has a *conclusion* represented by the formula to be proved and an empty set of *assumptions*, i.e., formulas that can be used in the proof of the conclusion. Firstly, the application of the `induction` tactic instantiates the variable l using an explicit induction schema issued from the recursive definition of **list** datatype. It produces two new subgoals under the form of logical implications whose conditions are shifted to the set of assumptions by the `intros` tactic. The proofs for each case are preceded by dash (-) symbols. The base case, when l is empty, can be proved by the `apply` tactic as an instance of the snil constructor of the **sorted**

definition. The proof of the step case starts by applying the `inversion` tactic on the assumption H[5] which is of the form **sorted** *(a :: a0 :: l)*. Since H can be generated only by the s2 constructor of **sorted**, the condition **sorted** *(a0 :: l)* of s2 should hold and can be added to the set of assumptions. On the other hand, **sorted** *(a0 :: l)* is also the conclusion of the goal, hence the proof can finish by the application of the `assumption` tactic.

Min-Max-Sort is certified if the following lemma is proved:

Lemma MMS_is_sound : is_a_sorting_algorithm MMS.

Its proof is based on the two main lemmas:

Lemma MMS_permutation : $\forall\ l$, permutation (MMS l) l.

Lemma MMS_is_sorted : $\forall\ l$, **sorted** (MMS l).

MMS_permutation is based on 6 (resp., 5) lemmas formalising crucial properties of count (resp., In), as well as 11 lemmas representing properties about minA, maxA, arithmetic and equality reasoning. On the other hand, MMS_is_sorted was based on 12 different lemmas, mostly related to properties of sorted. Most of lemmas have been proved using explicit induction with induction schemas built from the recursive definition of MMS (4 times, using the `Recdef` library and the `Functional Scheme` construction) and of the **list** datatype (21 times, using the `induction` tactic).

The main lemmas used to prove the properties about In are :

Lemma len_TrimMMA : $\forall\ x\ y\ l$, Nat.le (length (TrimMMA $x\ y\ l$)) (S (length l)).

Lemma In_TrimMMA : $\forall\ a\ b\ x\ l$, In x (TrimMMA $a\ b\ l$) $\rightarrow$ $a = x \vee b = x \vee$ In $x\ l$.

Lemma minMMA_id : $\forall\ x\ l$, minA $x\ l = x \vee$ maxA $x\ l = x \vee$ In x (TrimMMA $x\ x\ l$).

Lemma minMMA_id1 : $\forall\ a0\ x\ l$, $a0 < x \rightarrow$ (minA $a0\ l = x \vee$ maxA $x\ l = x \vee$ In x (TrimMMA $a0\ x\ l$)).

Lemma minMMA_id2 : $\forall\ b\ x\ l$, $x < b \rightarrow$ (minA $x\ l = x \vee$ maxA $b\ l = x \vee$ In x (TrimMMA $x\ b\ l$)).

Lemma minA_maxA_TrimMMA : $\forall\ l\ x\ a\ b$, In $x\ l \rightarrow$ **le** $a\ b \rightarrow$ (minA $a\ l = x \vee$ maxA $b\ l = x \vee$ In x (TrimMMA $a\ b\ l$)).

Lemma In_MMS : $\forall\ x\ l$, In $x\ l \rightarrow$ In x (MMS l).

Lemma minA_In: $\forall\ a\ x\ l$, minA $a\ l = x \rightarrow a = x \vee$ In $x\ l$.

Lemma maxA_In: $\forall\ a\ x\ l$, maxA $a\ l = x \rightarrow a = x \vee$ In $x\ l$.

Lemma MMS_In : $\forall\ x\ l$, In x (MMS l) $\rightarrow$ In $x\ l$.

[5] this assumption label was generated automatically by the Coq system.

The main lemmas about count are:

Lemma count_TrimMMA_TT : $\forall\ x\ a\ b\ l$, $(x$ =? minA $a\ l)$ = true $\rightarrow$ $(x$ =? maxA $b\ l)$ = true $\rightarrow$ $a \leq b \rightarrow$ S (S (count x (TrimMMA $a\ b\ l$))) = count x $(a :: b :: l)$.

Lemma count_TrimMMA_TF: $\forall\ x\ a\ b\ l$, $(x$ =? minA $a\ l)$ = true $\rightarrow$ $(x$ =? maxA $b\ l)$ = false $\rightarrow$ $a \leq b \rightarrow$ S (count x (TrimMMA $a\ b\ l$)) = count x $(a :: b :: l)$.

Lemma count_TrimMMA_FT: $\forall\ x\ a\ b\ l$, $(x$ =? minA $a\ l)$ = false $\rightarrow$ $(x$ =? maxA $b\ l)$ = true $\rightarrow$ $a \leq b \rightarrow$ S (count x (TrimMMA $a\ b\ l$)) = count x $(a :: b :: l)$.

Lemma count_TrimMMA_FF: $\forall\ x\ a\ b\ l$, $(x$ =? minA $a\ l)$ = false $\rightarrow$ $(x$ =? maxA $b\ l)$ = false $\rightarrow$ $a \leq b \rightarrow$ count x (TrimMMA $a\ b\ l$) = count x $(a :: b :: l)$.

Lemma count_MMS : $\forall\ x\ l$, count x (MMS l) = count $x\ l$.

The proofs of the lemmas about count are more complex as they involve arithmetic reasoning. These lemmas are not necessary in the case when the lists characterise sets (lists without repeating elements) for which a simpler permutation relation can be defined as:

Definition permutation_sets $l\ l' := \forall\ x$, In $x\ l \leftrightarrow$ In $x\ l'$.

Similarly, the lemmas about In can be ignored if the usual (weaker) definition of permutation on multisets is employed:

Definition permutation_multisets $l\ l' := \forall\ x$, count $x\ l$ = count $x\ l'$.

Finally, the lemmas used to prove MMS_is_sorted are:

Lemma minA_le_maxA: $\forall\ a\ b\ l$, $a \leq b \rightarrow$ minA $a\ l \leq$ maxA $b\ l$.

Lemma le_maxA : $\forall\ x\ l$, Nat.le x (maxA $x\ l$).

Lemma le_maxA_In : $\forall\ x\ y\ l$, In $y\ l \rightarrow$ Nat.le y (maxA $x\ l$).

Lemma MMS_max : $\forall\ x\ a\ b\ l$, In x (TrimMMA $a\ b\ l$) $\rightarrow$ Nat.le x (maxA a $(b :: l)$).

Lemma le_minA : $\forall\ x\ l$, Nat.le (minA $x\ l$) x.

Lemma le_minA_In : $\forall\ x\ y\ l$, In $y\ l \rightarrow$ Nat.le (minA $x\ l$) y.

Lemma MMS_min : $\forall\ x\ a\ b\ l$, In x (TrimMMA $a\ b\ l$) $\rightarrow$ Nat.le (minA a $(b :: l)$) x.

Lemma maxA_sym : $\forall\ a\ b\ l$, maxA a $(b :: l)$ = maxA b $(a :: l)$.

Lemma minA_sym : $\forall\ a\ b\ l$, minA a $(b :: l)$ = minA b $(a :: l)$.

Lemma sorted_sorted_rev: $\forall\ a\ l$, **sorted** $l \rightarrow$ $(\forall\ y$, In $y\ l \rightarrow a \leq y)$ $\rightarrow$ **sorted** $(a :: l)$.

Lemma sorted_app : $\forall\ y\ l$, $(\forall\ x$, In $x\ l \rightarrow$ Nat.le $x\ y)$ $\rightarrow$ **sorted** (l) $\rightarrow$ **sorted** $(l$ ++ $[y])$.

6 Conclusions

The main conclusions of this paper are based on the comparison between Coq and *Theorema* specifications and the proofs produced, as detailed below.

Specifications. The types in *Theorema* are not explicitly declared, but they are based on notation conventions. *Theorema* can accept partial functions, such as *Min* and *Max* that are not defined when the argument is an empty list. On the other hand, the Coq specifications are typed and accept only total functions. For the *Min-Max-Sort* example, the translation into Coq was straightforward since the call to these functions (that happens only in the definition of MMS) is done only when their arguments are non-empty lists and allowed us to replace them with the body of their definitions.

On the other hand, the Theorema specification for *Patience-Sort* uses partial functions, like head and *last*, that take lists as arguments. The solution used on the Coq side was to define the return value to be of type **option nat** such that when the arguments are empty lists the returned value is None, otherwise Some x, where x is the body of the partial function when called on non-empty lists. Below, we give the full Coq definitions of these functions, as well as of the auxiliary function *SelSort* that uses them:

```
Function SelSort (l: list nat):=
  match l with
    nil ⇒ Some nil
  | a :: l' ⇒
      match l' with
        nil ⇒ Some [a]
      | _ ⇒
          (match (SelSort (l')) with
             None ⇒ None
           | Some ss ⇒
               (match head (ss) with
                  None ⇒ None
                | Some hd ⇒ (match last ss with
                              None ⇒ None
                            | Some lst ⇒ if (a <=? hd) then Some (a :: ss)
                                         else if (lst <=? a)
                                            then Some (ss ++ [a])
                                            else Some ss
                            end)
                end
               )
          end)
      end
end.

Definition head (l: list nat):=
  match l with
  | nil ⇒ None
  | hd :: tl ⇒ Some hd
  end.
```

```
Function last (l: list nat) :=
  match l with
  | nil ⇒ None
  | hd :: tl ⇒ match tl with
                    nil ⇒ Some hd
                  | _ ⇒ last tl
                  end
  end.
```

Another important difference between systems is the following: in Coq one can only define terminating functions, while in *Theorema* this is not necessary.

Proofs. The proofs in both Coq and *Theorema* systems have been built using the *cascading* proof strategy: when a proof fails on some subgoal s, s is converted to a lemma whose proof may be based on other lemmas that are expected to be simpler to prove or are already proved. In Coq some lemmas can be directly accessed from the standard library that comes with the system, however, many necessary lemmas had to be created by the user. In *Theorema* for the certification process we do not need to use a specific domain of ordered elements, but we use inference rules which are based on the natural properties of total order (e.g. transitivity). The theory necessary for the proofs is not imported from libraries, but is constructed ad-hoc for the purpose of definition and certification of algorithms. However, we can also run the algorithms in *Theorema* and for this purpose we use numbers with the ordering provided by the *Mathematica* libraries.

During the design of the Coq proofs, we have used some lemmas from the List and Arith standard libraries, for example the Nat.lt_le_incl lemma that was proved in the file `Nat.v` from Arith. These libraries are not exhaustive and we found convenient to prove additional properties, for example:

```
Lemma leb_FT_eq : ∀ x a, (x <? a ) = false → (x <=? a ) = true → x = a.
```

```
Lemma leb_TT_eq : ∀ x a, (x <=? a ) = true → (a <=? x ) = true → x = a.
```

The *Theorema* proofs are automatically generated in natural style, are easy to read as they are similar to human proofs. In contrast, in Coq the user needs the computer to run the proof scripts step by step which displays the current state of the proof. The generation of Coq scripts was highly interactive. However, Coq gives the opportunity to the user to automate the proof process by using tactics, script-generators and solvers for decision problems.

In Coq, once the proofs are finished, they are certified by the kernel of the system using the Curry-Howard isomorphism, therefore the proofs are automatically ensured to be logically correct. In contrast, in *Theorema* it is possible to introduce inference rules that are not logically correct, therefore we took much care when designing them.

The *Theorema* certification of all the algorithms described in Section 3, as well as the Coq scripts can be found at https://members.loria.fr/SStratulat/files/SCSS2024.zip.

Future work. One important direction to proceed towards more efficient generation of such proofs is the automatic generation of the necessary lemmas in the cascading step.

This work constitutes a good basis for a possible integration of the two systems by translating the *Theorema* proofs into Coq scripts in order to be rigorously verified. Thus, we would have natural style proofs that use high-level inference rules and strategies, but can also be completely trusted because they are certified by a safe system.

Another interesting followup of this work in *Theorema* is the study of the duality between certification proofs and synthesis proofs.

Acknowledgements

This work is co-funded by the European Union, Erasmus+ project AiRobo: Artificial Intelligence based Robotics, 2023-1-RO01-KA220-HED-000152418.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P., Ulbrich, M.: Deductive Software Verification. The KeY Book: From Theory to Practice. Springer LNCS 10001 (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Baity, R., Humphrey, L.R., Hopkinson, K.: Formal Verification of a Merge Sort Algorithm in SPARK. AIAA SciTech Forum, American Institute of Aeronautics and Astronautics (2024/02/01 2021). https://doi.org/doi:10.2514/6.2021-0039, https://doi.org/10.2514/6.2021-0039
3. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving jdk's dual pivot quicksort correct. In: Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers 9. pp. 35–48. Springer (2017)
4. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, vol. XXV. Springer (2004)
5. Bockenek, J., Lammich, P., Nemouchi, Y., Wolff, B.: Using Isabelle/UTP for the verification of sorting algorithms: A case study. EasyChair Preprint no. 944 (2019). https://doi.org/10.29007/ddqm
6. Bove, A., Coquand, T.: Formalising bitonic sort in type theory. In: Types for Proofs and Programs (TYPES 2004). pp. 82–97. Springer LNCS 3839 (2004)
7. Buchberger, B.: Algorithm invention and verification by lazy thinking. In: Analele Universitatii de Vest, Timisoara, Ser. Matematica - Informatica. vol. XLI, pp. 41–70 (2003)
8. Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuta, E., Vasaru, D.: A survey on the Theorema project. In: In International Symposium on Symbolic and Algebraic Computation. pp. 384–391. ACM Press (1997)

9. Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., Windsteiger, W.: Theorema 2.0: Computer-Assisted Natural-Style Mathematics. Journal of Formalized Reasoning **9**(1), 149–185 (2016). https://doi.org/10.6092/issn.1972-5787/4568
10. Burgos, M.P.F.: Formalization of sorting algorithms in Isabelle/HOL. Master's thesis, Vrije Universiteit Amsterdam (2019)
11. 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)
12. Couturier, R.: Formal engineering of the bitonic sort using PVS. In: Proceedings of the 2nd Irish Conference on Formal Methods (IW-FM'98). p. 16–34. BCS Learning & Development Ltd., Swindon, GBR (1998)
13. Dramnesc, I., Jebelean, T.: Synthesis of List Algorithms by Mechanical Proving. Journal of Symbolic Computation **68**, 61–92 (2015). https://doi.org/10.1016/j.jsc.2014.09.030
14. Dramnesc, I., Jebelean, T.: Automatic synthesis of merging and inserting algorithms on binary trees using multisets in Theorema. In: International Conference on Mathematical Aspects of Computer and Information Science (MACIS 2019). pp. 153–168. Springer LNCS 11989 (2019)
15. Dramnesc, I., Jebelean, T.: Synthesis of sorting algorithms using multisets in Theorema. Journal of Logical and Algebraic Methods in Programming **119**, 100635 (2020). https://doi.org/10.1016/j.jlamp.2020.100635
16. Dramnesc, I., Jebelean, T.: AlCons: Deductive synthesis of sorting algorithms in Theorema. In: Theoretical Aspects of Computing (ICTAC 2021). pp. 314–333. Springer LNCS 12819 (2021). https://doi.org/10.1007/978-3-030-85315-0_18, https://doi.org/10.1007/978-3-030-85315-0_18
17. Dramnesc, I., Jebelean, T.: Mechanical verification of Insert-Sort and Merge-Sort using multisets in Theorema. In: IEEE 21st International Symposium on Intelligent Systems and Informatics (SISY 2023). pp. 55–60. IEEE Xplore (2023)
18. Dramnesc, I., Jebelean, T., Stratulat, S.: Mechanical synthesis of sorting algorithms for binary trees by logic and combinatorial techniques. Journal of Symbolic Computation **90**, 3–41 (2019)
19. Filliâtre, J.C., Magaud, N.: Certification of Sorting Algorithms in the System Coq. In: Theorem Proving in Higher Order Logics: Emerging Trends (1999)
20. Georgiou, P., Hajdu, M., Kovacs, L.: Saturating sorting without sorts. In: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2024). EPiC Series in Computing, vol. 100, pp. 88–105. EasyChair (2024). https://doi.org/10.29007/rg9z, https://easychair.org/publications/paper/qbDc
21. de Gouw, S., de Boer, F.S., Rot, J.: Verification of counting sort and radix sort. Deductive Software Verification–The KeY Book: From Theory to Practice pp. 609–618 (2016)
22. Hoang, D., Moy, Y., Wallenburg, A., Chapman, R.: Spark 2014 and GNATprove. International Journal on Software Tools for Technology Transfer **17**(6), 695–707 (2015). https://doi.org/10.1007/s10009-014-0322-5, https://doi.org/10.1007/s10009-014-0322-5
23. Howard, W.A.: The formulas-as-types notion of construction. In: To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, pp. 479–490. Academic Press (1980), reprint of 1969 article

24. 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
25. Kaufmann, M., Moore, J.S.: ACL2 Version 8.3 - The User's Manual (1999)
26. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley, 2 edn. (1998)
27. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. Journal of Logic and Algebraic Programming **58**(1-2), 89–106 (2004). https://doi.org/10.1016/j.jlap.2003.07.006, https://hal.science/hal-01984932
28. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Springer LNCS 2283 (2002)
29. Petrovic, D.: Verification of selection and heap sort using locales. Arch. Formal Proofs (2014), https://www.isa-afp.org/entries/Selection_Heap_Sort.shtml
30. Quarfot Orrevall, S., Gengelbach, A.: Implementation and Verification of Sorting Algorithms with the Interactive Theorem Prover HOL. Student thesis, Department of Information Technology, Mathematics and Computer Science, Disciplinary Domain of Science and Technology, Uppsala University (2020), http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-424295
31. Safari, M., Huisman, M.: A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In: Integrated Formal Methods. pp. 257–275. Springer LNCS 12546 (2020)
32. Sandip, R., Sumners, R.W.: Verification of an In-place Quicksort in ACL2. In: Proceedings of the 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (2002), https://api.semanticscholar.org/CorpusID:264243016
33. Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS prover guide - version 7.1. SRI International (November 2020)
34. Sternagel, C.: Proof pearl – a mechanized proof of GHCs mergesort. Journal of Automated Reasoning **51**(4), 357–370 (2013)
35. The Coq development team: The Coq Reference Manual. INRIA (2020), http://coq.inria.fr/doc
36. Tushkanova, E., Giorgetti, A., Kouchnarenko, O.: Specifying and proving a sorting algorithm. Tech. rep., Laboratoire d'Informatique de l'Université de Franche-Comte (2009)
37. Windsteiger, W.: Theorema 2.0: A system for mathematical theory exploration. In: ICMS'2014. LNCS, vol. 8592, pp. 49–52 (2014). https://doi.org/10.1007/978-3-662-44199-2_9
38. Zhang, Y., Zhao, Y., Sanan, D.: A verified timsort C implementation in Isabelle/HOL. arXiv preprint arXiv:1812.03318 (2018)