Certification of Tail Recursive Bubble–Sort in Theorema and Coq

Isabela Drămnesc¹, Tudor Jebelean², and Sorin Stratulat³

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

Algorithm certification or program verification have an increasing importance in the current technological landscape, due to the sharp increase in the complexity of software and software using systems and the high potential of adverse effects in case of failure. For instance robots constitute a particular class of systems that can present high risks of such failures. Sorting on the other hand has a growing area of applications, in particular the ones where organizing huge data collections is critical, as for instance in environmental applications.

We present an experiment in formal certification of an original version of the Bubble–Sort algorithm that is functional and tail recursive. The certification is performed in parallel both in Theorema and in Coq, this allows to compare the characteristics and the performance of the two systems. In Theorema the proofs are produced automatically in natural style (similar to human proofs), while in Coq they are based on scripts. However, the background theory, the algorithms, and the proof rules in Theorema are composed by the user without any restrictions – thus error prone, while in Coq one can only use the theories and the proof rules that are rigorously checked by the system, and the algorithms are checked for termination.

The goal of our experiments is to contribute to a better understanding and estimation of the complexity of such certification tasks and to create a basis for further increase of the level of automation in the two systems and for their possible integration.

1 Introduction

Sorting algorithms are essential in a variety of computational activities, forming the foundation for numerous applications across different fields, especially those concerning the environment, climate change, and more. This is because of the considerable amount of data that needs to be organized and handled efficiently. With data size and complexity increasing rapidly, the effectiveness and accuracy of sorting algorithms are more crucial than ever. A particularly
important application of program verification is the area of robotics. In this area there are various important aspects as human interaction, robot interaction, possibility of accidents, etc. that make formal certification of most robotic algorithms absolutely necessary.

In this paper we focus on the certification of the tail recursive version of the Bubble-Sort algorithm that was firstly introduced in [8]. Initially, the authors applied some special techniques for the automated synthesis from proofs of the algorithm Max-Sort and the corresponding auxiliary functions. The use of the two auxiliary functions max (that extracts the maximum from a list), and Trim (that returns the list without the maximum element) is inefficient as the scan of the list is performed twice. For efficiency, the authors transformed these two functions into one single function maxTrim that returns the maximum and the list without it. Then, the tail recursive sorting algorithm that uses the function maxTrim leads in fact to the tail recursive version of Bubble-Sort. The definition of these algorithms is given in Theorema and in Coq in Section 2. Other versions of Bubble-Sort (with a flag, functional and imperative) are also derived in [8], however in this paper the authors focus on the certification of the tail recursive version together with the auxiliary function.

In Theorema the background theory, the algorithms, and the proof rules are composed ad-hoc by the user without any restrictions, therefore they are error prone. In Coq, in contrast, one can only use the theories and the proof rules that are rigorously checked by the system, and the algorithms are checked for termination. Therefore it is very useful to perform the certification in both systems, thus benefiting both from the natural style of Theorema and by the rigor of Coq.

The Theorema system[4, 5, 25] is a framework built upon Mathematica\textsuperscript{1} that supports the processes of defining mathematical theories, including definition of algorithms by logical formulae, experimenting by running the algorithms, and developing and using mechanical provers. The system facilitates the certification of algorithms because their implementation in Theorema does not use a programming language, they being defined directly in predicate logic together with their specification. A distinctive feature of the Theorema system is the use of natural style (similar to human) for expressing the logical formulae and the algorithms, for the inference rules of the provers, and for the presentation of the proofs.

The Coq system [23, 1] is a skeptical proof assistant widely used to certify algorithms. The certification procedure is based on the Curry-Howard isomorphism [17], where the proofs are interpreted as programs/terms and formulas as types, and allows to check if, given a proof p of a formula f, the type of the p term is of type f.

The proofs in the Theorema system are generated automatically, without the need of human interaction, 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 and to display the current state of the proof.

Related work. Classical algorithms on arrays/lists [16, 24, 20, 26, 6, 3, 21] have been certified in formal certification environments like Coq [1] and Isabelle/HOL [19].

[14] proves the correctness of various sorting algorithms using the Why3 [15] platform. In [16], the authors verify three imperative sorting algorithms, insertion sort, quick sort and heap sort, in Coq. To prove the permutation property, they propose to express that the set of permutations is the smallest equivalence relation containing the transpositions (i.e., the exchanges of elements). In [22], the authors follow this approach to formally define permutation and they introduce a generic pattern to verify the permutation property of bubble sort, selection
sort, insertion sort, parallel odd-even transposition sort, quick sort, two in-place merge sorts and TimSort for any arbitrary size of input using VerCors [2].

Although there is much literature on the verification of sorting algorithms, none of the approaches uses natural style proving (except our work on verification and synthesis that we summarize below). The algorithms Insert–Sort and Merge–Sort have been formally verified by the authors in the Theorema system in [11]. The automated certification in both Theorema and Coq of the sorting algorithms: Quick–Sort, Patience–Sort, Min–Sort, Max–Sort, Min–Max–Sort is submitted for review to [13]. These algorithms have been synthesized in authors’ previous research, see [12, 10, 7, 9]. The tail recursive version of Bubble–Sort that is certified in this paper, was firstly introduced in [8].

Also, there is no previous work neither in Coq nor in Theorema on the verification of the algorithms presented here.

The novelty/contribution of the paper consists in:

- the first certification of the tail recursive version of Bubble–Sort in Theorema and Coq;
- the comparison of the two systems on a similar task;
- the use in Theorema of multisets in order to express the fact that lists have same elements and to simplify the proofs related to this.

2 Notations and algorithms

2.1 Notations in Theorema

We consider multisets and lists over a totally ordered domain and we use uppercase roman letters for lists (U, V, T). ⟨⟩ is the empty list, and in the form head–tail this is denoted by a ◁ U, (a is the head and U is the tail of the list – which is always a list). Lowercase roman letters like a, b, c, x denote the elements of lists or multisets. These are objects from a totally ordered domain (notation < and ≤). ∅ denotes the empty set, {a} is the multiset containing the element a with multiplicity one, and M[U] denotes the multiset of the list U. ψ is the additive union of multisets (keeps the multiplicity of elements), like in [18]. The total ordering between the elements is extended also between an element and a list (el ≤ U denotes that the element el is smaller or equal to each element of the list U, U ≤ el denotes that each member of the list U are smaller or equal to the element el); and between two lists (U ≤ V denotes that each member of U is smaller or equal to each element of V).

The type of the objects is not used explicitly, but this is automatically detected by the prover according to the notations and depending on the context in which the objects occur.

In Theorema, for function and predicate application we use squared brackets (e.g., F[x], P[x]). Quantified variables are written under the quantifier (e.g. ∀ “for all X”, ∃ “exists X”), and Skolem constants have integer indices (e.g., U₀, a₀).

Basic definitions.

Definition 1. ∀ a,u. (IsSorted[⟨⟩] \iff (a ≤ U ∧ IsSorted[U]))

Definition 2. ∀ a,u. (M[⟨⟩] = ∅ \iff M[a ◁ U] = {a} \uplus M[U])
Tail recursive Bubble–Sort in *Theorema*. In [8] the authors synthesized the Max-Sort algorithm together with the auxiliary functions max and Trimm. For efficiency, by applying some transformation rules, the authors derived the tail recursive versions of these algorithms, resulting in the ones presented below.

The use of the two functions max and Trimm together is quite inefficient because the scan of the list is performed twice, using the same test at each step. Therefore, the two functions are merged into one, which returns the pair of maximum and the list without it:

**Algorithm 1.** Tail recursive max and Trimm.

\[
\forall a, b, U, V
\begin{aligned}
\maxTrimm[a \sim U] &= \maxTrA[U, a, \emptyset] \\
\maxTrA[\emptyset, a, V] &= (V, a) \\
\maxTrA[b \sim U, a, V] &=
\begin{cases}
\maxTrA[U, b, V \setminus a], & \text{if } a \leq b \\
\maxTrA[U, a, V \setminus b], & \text{if } b < a
\end{cases}
\end{aligned}
\]

The nontrivial branch of the sorting algorithm is expressed in the following way, also as a tail recursive function, which is in fact the algorithm Bubble-Sort:

**Algorithm 2.** Bubble-Sort.

\[
\forall a, b, U, V
\begin{aligned}
\text{BSort}[a \sim U] &= \text{BSortA}[\maxTrA[U, a, \emptyset], \emptyset] \\
\text{BSortA}[(\emptyset, a), V] &= a \sim V \\
\text{BSortA}[b \sim U, a, V] &= \text{BSortA}[\maxTrA[U, b, \emptyset], a \sim V]
\end{aligned}
\]

This algorithm is known as its more efficient version which finishes as soon as the list is already sorted.

2.2 Notations in Coq

In the Coq script, the multisets manipulated by the sorting algorithm are represented as lists of naturals, of type *list nat*. The constructors for *list* are *nil* and ::, and for *nat* are 0 and S. The operations on multisets can be reproduced via a permutation relation on lists, based on the built-in In predicate and the used-defined count function, as explained in the 'Basic definitions' paragraph. The predicates are defined inductively, using the *Inductive* keyword.

In Coq, any recursive function should be total and terminating. The totality can be syntactically checked if the function definition uses match constructs to detail its behavior according to the values that some (matching) expression e, usually one of the function arguments, can take by using the constructors of the type of e. The termination property requires that some function argument should decrease (w.r.t. some well-founded order) after each recursive call. Coq uses the *Fixpoint* keyword for defining the recursive functions for which Coq automatically identifies the recursive function argument and some subterm (syntactic) well-founded order. The keyword *Function* is used when the well-founded order is explicitly defined using the *wf* keyword.

**Basic definitions.** The definitions for the sorting and permutation predicates are:

```coq
Inductive IsSorted : list nat -> Prop :=
| snil : IsSorted nil | s1 : \forall x, IsSorted (x::nil) | s2 : \forall x y l, IsSorted (y::l) \to x \leq y \to IsSorted (x::y::l).
```

56
Definition  
permutation \( l \ l' := \)
\[
\forall x, (\text{In } x \ l \leftrightarrow \text{In } x \ l') \land \text{count } x \ l = \text{count } x \ l'.
\]
where the \textit{count} function is defined as:

\[
\text{Fixpoint count } x \ l := \\
\text{match } l \text{ with} \\
\text{nil } \Rightarrow 0 \\
| \text{hd :: tl } \Rightarrow \text{if } x =? \text{hd} \text{ then } S (\text{count } x \ tl) \text{ else count } x \ tl \\
\text{end.}
\]

The \textit{=}? notation represents the boolean equality that helps to compare two naturals.

Tail recursive Bubble–Sort in Coq. The Coq definitions for the auxiliary \textit{maxTrA} and \textit{BSortA} recursive functions from \textit{Theorema} are:

\[
\text{Fixpoint maxTrA } l \ a \ V := \\
\text{match } l \text{ with} \\
\text{nil } \Rightarrow (V, a) \\
| b :: U \Rightarrow \text{if leb a b then maxTrA } U \ b (V ++ \ [a]) \text{ else maxTrA } U \ a (V ++ \ [b]) \\
\text{end.}
\]

Function \textit{BSortA} \( p \ {\text{wf (fun } p1 \ p2 \Rightarrow}} \)
\[
\text{Nat.lt (length (fst (fst p1))) (length ((fst (fst p2))))) } p):=
\]

\[
\text{match } p \text{ with} \\
((\text{nil, a}), V) \Rightarrow a :: V \\
| ((b :: U, a), V) \Rightarrow \text{BSortA } ((\text{maxTrA } U \ b \ \text{nil}), (a :: V)) \\
\text{end.}
\]

The \textit{leb} function returns the boolean result of the 'less or equal' comparison between the two naturals given as arguments, while \textit{Nat.lt} is the inductive predicate 'less than' over naturals. The \textit{length} function returns the length of a list and ++ is the concatenation operator on lists.

Contrary to the \textit{Theorema} notation, \textit{BSortA} takes only one argument, which is the pair of the first and second arguments used for the \textit{Theorema} notation. The function \textit{fst} (resp., \textit{snd}) returns the first (resp., second) element of a pair.

Finally, \textit{BSort} is defined as:

\[
\text{Definition BSort } l := \\
\text{match } l \text{ with} \\
\text{nil } \Rightarrow \text{nil} \\
| a :: U \Rightarrow \text{BSortA } ((\text{maxTrA } U \ a \ \text{nil}), \text{nil}) \\
\text{end.}
\]

3 Verification in \textit{Theorema}

For the verification of the sorting algorithm \textit{BSort} we have to prove the following two theorems: that the algorithm preserves multisets (Theorem 1) and that the output is sorted (Theorem 2).

Theorem 1. \( \forall X \left( \mathcal{M}[X] = \mathcal{M}[\text{BSort}[X]] \right) \)

Theorem 2. \( \forall X \left( \text{IsSorted}[\text{BSort}[X]] \right) \)
3.1 Proof of Theorem 1

For proving Theorem 1 we consider the following properties in the knowledge base: \( \cup \) is associative, is commutative, has unit; \( \leq \) is transitive, for any \( X : X \leq \langle \rangle, \langle \rangle \leq X \); and:

**Property 1.** \( \forall a,b,X (a \sim X \leq b) \iff (X \leq b \land a \leq b) \)

**Property 2.** \( \forall a,b,X (b \leq a \sim X) \iff (b \leq X \land b \leq a) \)

**Property 3.** \( \forall a,X (M[X \sim a] = M[X] \cup \{a\}) \)

**Property 4.** \( \forall X,Y (M[X \bowtie Y] = M[X] \cup M[Y]) \)

**Property 5.** \( \forall a,X (M[\langle X, a \rangle] = M[X] \cup \{a\}) \)

**Property 6.** \( \forall X,Y (M[\langle X, Y \rangle] = M[X] \cup M[Y]) \)

**Property 7.** \( \forall U,V (M[maxTrA[U, a, V]] = M[U] \cup \{a\} \cup M[V]) \)

**Property 8.** \( \forall U,V (M[BSortA[U, V]] = M[U] \cup M[V]) \)

*Proof.* Take \( a_0, U_0 \) arbitrary, but fixed, and according to the Algorithm 2 prove:

\[ M[a_0 \sim U_0] = M[\text{Bubble-Sort}[a_0 \sim U_0]] \]  \hspace{1cm} (1)

which becomes

\[ M[a_0 \sim U_0] = M[\text{BSortA}[maxTrA[U_0, a_0, \langle \rangle], \langle \rangle]] \]  \hspace{1cm} (2)

By Definition 2 the goal becomes:

\[ \{a_0\} \cup M[U_0] = M[\text{BSortA}[maxTrA[U_0, a_0, \langle \rangle], \langle \rangle]] \]  \hspace{1cm} (3)

By Property 8 the goal becomes:

\[ \{a_0\} \cup M[U_0] = M[maxTrA[U_0, a_0, \langle \rangle]] \cup M[\langle \rangle] \]  \hspace{1cm} (4)

By Definition 2 and by union properties, the goal becomes:

\[ \{a_0\} \cup M[U_0] = M[maxTrA[U_0, a_0, \langle \rangle]] \]  \hspace{1cm} (5)

By Property 7 the goal becomes:

\[ \{a_0\} \cup M[U_0] = M[U_0] \cup \{a_0\} \cup M[\langle \rangle] \]  \hspace{1cm} (6)

This holds by Definition 2 and by properties of multiset union. \( \Box \)
The proof of Property 7.

Proof. Prove $\forall a,U,V \left( M[\text{maxTrA}[U,a,V]] = M[U] \uplus \{a\} \uplus M[V] \right)$

by induction on $U$:

Base case: Take $a_0, V_0$ arbitrary but fixed and prove

$M[\text{maxTrA}[\{\}, a_0, V_0]] = M[\{\}] \uplus \{a_0\} \uplus M[V]$ (7)

By Definition 2 and by union properties, the goal becomes:

$M[\{a_0\}] = \{a_0\} \uplus M[V]$ (8)

By Algorithm 1 the goal becomes:

$M[\langle a_0 \rangle] = \{a_0\} \uplus M[V]$ (9)

By Property 5 the goal becomes:

$M[V_0] \uplus \{a_0\} = \{a_0\} \uplus M[V]$ (10)

This holds by the commutativity of $\uplus$.

Inductive step: Take $b_0, U_0$ arbitrary but fixed $(a, V$ remains universally quantified), assume:

$\forall a,V \left( M[\text{maxTrA}[U_0,a,V]] = M[U_0] \uplus \{a\} \uplus M[V] \right)$ (11)

and prove

$\forall a,V \left( M[\text{maxTrA}[b_0 \sim U_0,a,V]] = M[b_0 \sim U_0] \uplus \{a\} \uplus M[V] \right)$ (12)

We take $a_0, V_0$ arbitrary but fixed. By Definition 2 the goal becomes:

$M[\text{maxTrA}[b_0 \sim U_0,a_0,V_0]] = \{b_0\} \uplus M[U_0] \uplus \{a_0\} \uplus M[V_0]$ (13)

We prove (13) by cases using Algorithm 1:

Case 1: $a_0 \leq b_0$. The goal becomes:

$M[\text{maxTrA}[U_0,b_0 \sim a_0,V_0]] = \{b_0\} \uplus M[U_0] \uplus \{a_0\} \uplus M[V_0]$ (14)

By induction hypothesis (11) the goal becomes:

$M[U_0] \uplus \{b_0\} \uplus M[V_0 \sim a_0] = \{b_0\} \uplus M[U_0] \uplus \{a_0\} \uplus M[V_0]$ (15)

This holds by properties of multiset union.

Case 2: $b_0 < a_0$. The proof is analogous to the previous case.

The proof of Property 8.

Proof. According to Algorithm 2 the first argument of $\text{BSortA}$ always consists of a pair between

a list and an element. Therefore it is sufficient to show:

$\forall a,U,V \left( M[\text{BSortA}[U,a,V]] = M[U] \uplus \{a\} \uplus M[V] \right)$ (16)
Take \(a_0, V_0\) arbitrary, but fixed and prove by induction on the multilist (with respect to strict inclusion) of the first argument of \(BSortA\):

\[
\forall U \left( M[BSortA[(U, a_0), V_0]] = M[U] \uplus \{a_0\} \uplus M[V_0] \right)
\]  

(17)

**Base case:** Prove

\[
M[BSortA[(\langle\rangle, a_0), V_0]] = M[\langle\rangle] \uplus \{a_0\} \uplus M[V_0]
\]

(18)

By Definition 2 and by property unit the goal becomes:

\[
M[BSortA[(\langle\rangle, a_0), V_0]] = \{\{a_0\}\} \uplus M[V_0]
\]

(19)

By Algorithm 2 the goal becomes:

\[
M[a_0 
\triangle V_0] = \{\{a_0\}\} \uplus M[V_0]
\]

(20)

By Definition 2 the goal becomes:

\[
\{\{a_0\}\} \uplus M[V_0] = \{\{a_0\}\} \uplus M[V_0]
\]

(21)

This holds by reflexivity of equality.

**Induction step:** Take \(b_0, U_0\) arbitrary, but fixed and prove

\[
M[BSortA[(b_0 \triangle U_0, a_0), V_0]] = M[b_0 \triangle U_0] \uplus \{a_0\} \uplus M[V_0]
\]

(22)

By Definition 2 the goal becomes:

\[
M[BSortA[(b_0 \triangle U_0, a_0), V_0]] = \{\{b_0\}\} \uplus M[U_0] \uplus \{\{a_0\}\} \uplus M[V_0]
\]

(23)

By Algorithm 2 the goal becomes:

\[
M[BSortA[maxTrA[U_0, b_0, \langle\rangle], V_0] = \{\{b_0\}\} \uplus M[U_0] \uplus \{\{a_0\}\} \uplus M[V_0]
\]

(24)

Since \(maxTrA\) preserves multisets, the first argument of \(BSortA\) in the current goal is strictly included in the first argument of the inductive goal, thus by generalized induction the goal becomes:

\[
M[maxTrA[U_0, b_0, \langle\rangle]] \uplus M[a_0 \triangle V_0] = \{\{b_0\}\} \uplus M[U_0] \uplus \{\{a_0\}\} \uplus M[V_0]
\]

(25)

By Property 7 the goal becomes:

\[
M[U_0] \uplus \{\{b_0\}\} \uplus M[\langle\rangle] \uplus M[a_0 \triangle V_0] = \{\{b_0\}\} \uplus M[U_0] \uplus \{\{a_0\}\} \uplus M[V_0]
\]

(26)

This holds by Definition 2, and by the properties of multiset union. □

### 3.2 Proof of Theorem 2

For proving \(\forall X \left( IsSorted[BSort[X]] \right)\) we need certain properties among the current arguments of \(maxTrA\), namely when we have a call \(maxTrA[U, a, V]\), then \(V \leq a\). In order to reason about the arguments of \(maxTrA\) we define the relation \(E\) that describes the evolution of the arguments and the corresponding property \(P\).
Definition 3.
\[
\forall_{G,H} (E[G,H] \iff \left( \exists_{a,b,U,V} (G = \langle b \sim U, a, V \rangle \land (a \leq b) \land (H = \langle U, b, V \sim a \rangle)) \lor \\
(b < a \land H = \langle U, a, V \sim b \rangle)) \right) \\
\forall_{G,H} (E^*[G,H] \iff (G = H \lor (\exists_{K} (E^*[G,K] \land E[K,H]))))
\]

Definition 4.
\[
\forall_{G} (P[G] \iff \left( \exists_{a,U,V} (G = \langle U, a, V \rangle \land V \leq a) \right)
\]

The following is an elementary consequence of this definition:

Property 9. \forall_{a,U,V} (P[\langle U, a, V \rangle] = \Rightarrow V \leq a)

We also need the following properties in the knowledge base:

Property 10. \forall_{a,b,L} (b \leq a \land L \leq a \iff b \sim L \leq a)

Property 11. \forall_{a,b,L} (a \leq b \land V \leq a \iff V \sim a \leq b)

Property 12. \forall_{x,y \neq \langle \rangle, z} (x \leq y \land y \leq z \Rightarrow x \leq z)

Property 13. \forall_{G,H} (E[G,H] \Rightarrow (P[G] \Rightarrow P[H]))

Proof. We take arbitrary but fixed \(G, H\) and we assume \(E[G,H]\), whose existential definition allows us to find \(a_0,b_0,U_0,V_0\) such that:

\[G = \langle b_0 \sim U_0, a_0, V_0 \rangle\] (27)

and

\[(a_0 \leq b_0 \land (H = \langle U_0, b_0, V_0 \sim a_0 \rangle)) \lor (b_0 < a_0 \land H = \langle U_0, a_0, V_0 \sim b_0 \rangle)\] (28)

We also assume \(P[G]\) which by (27) becomes:

This property is the most important because it shows that the evolution relation transports the property.

\[P[\langle b_0 \sim U_0, a_0, V_0 \rangle]\] (29)

From this by (9) we obtain

\[V_0 \leq a_0\] (30)

In order to prove \(P[H]\), by the definition of \(P\), we prove:

\[\exists_{a,U,V} (H = \langle U, a, V \rangle \land V \leq a)\] (31)

We take \(U = U_0\) and prove:

\[\exists_{a,V} (H = \langle U_0, a \rangle \land V \leq a)\]

by cases using the disjunction (28).
The proof of Property 14 is straightforward by Definition 3, and by Property 13.

**Property 15.** \( \forall_{a,b,U,V,W} \left( E^*[U,a,\langle \rangle, V,b, V] \implies V \leq b \right) \)

The proof of Property 15 uses Definition 4, and the property \( a \leq \langle \rangle \).

The algorithm \( \text{maxTrA} \) terminates because the only recursive calls from Algorithm 1 reduce the argument \( b \sim U \) to \( U \). On the other hand the only terminating definition is the base case of Algorithm 1, and this gives the result in the form \( \langle V, a \rangle \). From this and Property 15 we can infer:

**Property 16.** \( \forall_{b, U,a,V} \left( \text{maxTrA}[U, b, \langle \rangle] = \langle V, a \rangle \land V \leq a \right) \)

In order to prove that \( \text{Bubble-Sort} \) returns a sorted list we need a certain property among the current arguments of \( \text{BSortA} \), namely when we have a call \( \text{BSortA}[\langle U, a \rangle, V] \), then \( a \leq V \) and \( V \) is sorted. In order to reason about the arguments of \( \text{BSortA} \) we define the relations \( \text{Es} \) and \( \text{Es}^* \) that describe the evolution of the arguments and the corresponding property \( \text{Ps} \).

**Definition 5.**
\[
\begin{align*}
\forall_{G,H} \left( \text{Es}[G,H] \iff \exists_{a,b,U,V} \left( G = \langle b \sim U, a \rangle, V \land H = \langle \text{maxTrA}[U,b,\langle \rangle, a \sim V] \rangle \right) \right) \\
\forall_{G,H} \left( \text{Es}^*[G,H] \iff \left( G = H \lor \exists_{K} \left( \text{Es}^*[G,K] \land \text{Es}^*[K,H] \right) \right) \right)
\end{align*}
\]

**Definition 6.** \( \forall_{G} \left( \text{Ps}[G] \iff \exists_{a,U,V} \left( G = \langle \langle U, a \rangle, V \rangle \land \text{IsSorted}[a \sim V] \land U \leq V \right) \right) \)

The following is an elementary consequence of this definition:

**Property 17.** \( \forall_{a,U,V} \left( \text{Ps}[\langle \langle U, a \rangle, V \rangle] \implies \text{IsSorted}[a \sim V] \land U \leq a \sim V) \right) \)

We prove now that the evolution relation transports the property.

**Property 18.** \( \forall_{G,H} \left( \text{Es}[G,H] \implies (\text{Ps}[G] \implies \text{Ps}[H]) \right) \)

**Proof.** The proof of Property 18 applies general inference rules and by Algorithm 2 the goal is
\[
\exists_{a,b,U,V} \left( G_0 = \langle b \sim U, a \rangle, V \rangle \land H_0 = \langle \text{maxTrA}[U,b,\langle \rangle, a \sim V] \rangle \right) \quad (32)
\]
By the above we can find \( a_0, b_0, U_0, V_0 \) such that
\[
G_0 = \langle b_0 \sim U, a_0 \rangle, V_0 \rangle \quad (33)
\]
and
\[
H_0 = \langle \text{maxTrA}[U_0,b_0,\langle \rangle, a \sim V_0] \rangle \quad (34)
\]
From the previous assumption by (33) we obtain

\[ Ps[(b_0 \sim U_0, a_0), V_0] \]  \hspace{1cm} (35)

From this by Property 17 we obtain \( IsSorted[a_0 \sim V_0] \) and \( b_0 \sim U_0 \leq a_0 \sim V_0 \). In order to prove \( Ps[H_0] \) by Definition 6 we prove

\[ \exists a, U, V \ (H_0 = ((U, a), V) \wedge IsSorted[a \sim V] \wedge U \leq a \sim V) \]  \hspace{1cm} (36)

We instantiate Property 16 with \( a \rightarrow b_0 \) and \( U \rightarrow U_0 \) and take \( V_1, b_1 \) such as:

\[ maxTrA[U_0, b_0, ()] = (V_1, b_1) \]  \hspace{1cm} (37)

and

\[ V_1 \leq b_1 \]  \hspace{1cm} (38)

By applying Definition 2, Properties 7, 5, and multisets preserve ordering we obtain:

\[ b_1 \leq a_0 \sim V_0, \text{ and } V_1 \leq b_1 \sim (a_0 \sim V_0) \]  \hspace{1cm} (39)

In order to prove (36) we take \( U \rightarrow V_1, a \rightarrow b_1 \) and \( V \rightarrow b_1 \sim (a_0 \sim V_0) \), and by (39) the goal reduces to:

\[ IsSorted[b_1 \sim (a_0 \sim V_0)] \wedge V_1 \leq b_1 \sim (a_0 \sim V_0) \]  \hspace{1cm} (40)

which by Definition 1 and our assumptions derived so far reduces to (38).

\[ \square \]

**Property 19.** \( \forall G, H \ (Es^*[G, H] \implies (Ps[G] \implies Ps[H])) \)

The proof is straightforward, by cases using Definition 5 and Property 18.

**Property 20.** \( \forall a, b, U, V \ (Es^*[\maxTrA[U, a, ()], ()], (), b), V] \implies IsSorted[b \sim V]) \)

**Proof.** Take \( a_0, b_0, U_0, V_0 \) arbitrary but fixed. Assume

\[ Es^*[\maxTrA[U_0, a_0, ()], (), (), b_0)], V_0]) \]  \hspace{1cm} (41)

and prove

\[ IsSorted[b_0 \sim V_0] \]  \hspace{1cm} (42)

(41) by Property 19 becomes:

\[ Ps[\maxTrA[U_0, a_0, ()], ()] \implies Ps[\langle (), b_0 \rangle, V_0]) \]  \hspace{1cm} (43)

First we prove the left hand side of (43) by using Property 16, and Definitions 4,1. From this we know

\[ Ps[\langle (), b_0 \rangle, V_0]) \]  \hspace{1cm} (44)

From (44) by Definition 6 we obtain \( IsSorted[b_0 \sim V_0] \wedge () \leq b_0 \sim V_0 \) which proves our goal. \( \square \)
The algorithm \( BSortA \) terminates because the only recursive call of the Algorithms 2 reduces the argument \( \langle b \sim U, a \rangle \) to \( maxTrA[U, b, \langle \rangle \] . This follows from Property 7 and to properties of multisets:

\[
\mathcal{M}(maxTrA[U, b, \langle \rangle ] = \mathcal{M}[U] \cup \{\{b\}\} \text{ is strictly included in } \mathcal{M}[U] \cup \{\{a, b\}\} = \mathcal{M}[\langle b \sim U, a \rangle ]
\]

On the other hand the only terminating definition is the first from Algorithm 2. This has as argument the shape specified in Property 20 as final configuration, and it gives the result in the form \( a \sim V \) which by Property 20 has the property \( \text{IsSorted}(b \sim V) \).

From this follows easily:

**Property 21.** \( \forall a, U \exists b, V (BSortA[maxTrA[U, a, \langle \rangle ], \langle \rangle ] = b \sim V \land \text{IsSorted}(b \sim V)]) \)

The proof of Theorem 2 follows from Property 21 and the Algorithm 2.

## 4 Certification in Coq

The \( BSort \) algorithm can be certified by proving the following main theorem:

**Lemma** \( BS\_is\_sound : \text{is\_a\_sorting\_algorithm} \ BSort. \)

where \( \text{is\_a\_sorting\_algorithm} \) is the function used to check the soundness property to be satisfied by the sorting function \( f \) given as argument:

**Definition** \( \text{is\_a\_sorting\_algorithm} (f : \text{list nat} \to \text{list nat}) := \forall al, \text{permutation} (f \ al) \ al \land \text{IsSorted} (f \ al). \)

In line with the results from [22], we certified only the ‘permutation’ property, based on the following two (‘In’ and ‘count’) lemmas:

**Lemma** \( BS\_in\_equiv: \forall x l, \text{In} x l \leftrightarrow \text{In} x (BSort l). \)

**Lemma** \( BSort\_count : \forall x l, \text{count} x l = \text{count} x (BSort l). \)

The proof of the \( BS\_in\_equiv \) lemma is based on the following lemmas:

**Lemma** \( BSortA\_in\_rev : \forall x p, \text{In} x (BSortA p) \to (\text{snd} (f \ p)) = x \lor \text{In} x (f \ (f \ p)) \lor \text{In} x (\text{snd} p). \)

**Lemma** \( BS\_in\_rev : \forall x l, \text{In} x (BSort l) \to \text{In} x l. \)

**Lemma** \( maxTrA\_in\_fst : \forall U a b L, \text{In} b L \to \text{In} b (maxTrA U a L). \)

**Lemma** \( maxTrA\_in : \forall U x b L, \text{In} x (U++L++[b]) \to \text{snd} (maxTrA U b L) = x \lor \text{In} x (f \ (f \ L)). \)

**Lemma** \( BSortA\_in : \forall x p, (\text{snd} (f \ p)) = x \lor \text{In} x (f \ (f \ p)) \lor \text{In} x (\text{snd} p) \to \text{In} x (BSortA p). \)

**Lemma** \( BS\_in : \forall x l, \text{In} x l \to \text{In} x (BSort l). \)

The proof of the \( BSort\_count \) lemma is based on the lemmas:

**Lemma** \( count\_app : \forall x l1 l2, \text{count} x (\text{app} l1 l2) = \text{count} x l1 + (\text{count} x l2). \)

**Lemma** \( count\_maxTrA : \forall x U b L, \text{count} x (U++L++[b]) = \\
(\text{if} x =? \text{snd} (maxTrA U b L) \text{then} 1 \text{ else} 0) + \text{count} x (f \ (f \ U b L)). \)
Lemma BSortA_count : \forall x p, count x (BSortA p) =
((if x =? (snd (fst p)) then 1 else 0) + count x (snd p)) + (count x (fst (fst p))).

Lemma BSortA_count : \forall x p, count x (BSortA p) =
((if x =? (snd (fst p)) then 1 else 0) + count x (snd p)) + (count x (fst (fst p))).

Most of lemmas have been proved using explicit induction, using

• the induction tactic (7 times), based on induction schemas issued from the inductive
definitions of the list datatype, and
• the functional induction tactic (3 times), based on induction schemas resulting from
the recursive definition of BSortA and implemented using the RecDef library and the
Functional Scheme construction.

The proofs of the four 'count'-related lemmas are more complex as they involve arithmetic
reasoning. It can be noticed that simpler proofs can obtained if the lists represent sets instead
of multisets, for which a simpler permutation relation can be defined as:

Definition permutation l l' := \forall x, (In x l \leftrightarrow In x l').

Also, the 'In'-related lemmas are useless if the following (weaker) definition of permutation
on multisets is employed instead:

Definition permutation l l' := \forall x, count x l = count x l'.

The certification of the 'sorting' property was more involved. It required the two-parameter
(non tail-recursive) version of maxTrA, referred to as maxTrN and defined as:

Fixpoint maxTrN l a :=
match l with
| nil => ([], a)
| b :: U => if leb a b
then ((a :: (fst (maxTrN U b))), Nat.max b (snd (maxTrN U b)))
else ((b :: (fst (maxTrN U a))), Nat.max a (snd (maxTrN U a)))
end.

We have shown its equivalence with maxTrA:

Lemma maxTrN_maxTrA_nil : \forall l a, maxTrA l a [] = maxTrN l a.

The equivalence proof was based on the following two lemmas:

Lemma maxTrN_max : \forall l a, snd (maxTrN l a) = list_max (a :: l).

Lemma maxTrN_maxTrA : \forall l a U, maxTrA l a U = (U ++ fst (maxTrN l a),
snd (maxTrN l a)).

The crucial lemma for proving the 'sorting' property is:

Lemma BSort_is_sorted' : \forall n l U n1, (\forall x, In x (n1 :: l) \rightarrow IsSorted (x :: U)) \rightarrow
length l = n \rightarrow IsSorted (BSortA ((maxTrA l n1 []), U)).

Instead, we have proved the BSort_is_sorted lemma:

Lemma BSort_is_sorted : \forall n l U n1, (\forall x, In x (n1 :: l) \rightarrow IsSorted (x :: U)) \rightarrow
length l = n \rightarrow IsSorted (BSortA ((maxTrN l n1), U)).
which resulted from the replacement of \((\text{maxTrA} \ l \ n1 \ []\)) by \((\text{maxTrN} \ l \ n1)\). Finally, in the proof of the 'sorting' property, we have replaced \text{maxTrA} by \text{maxTrN} using the \text{maxTrN_maxTrA_nil} lemma before calling \text{BSort_is_sorted}.

Other lemmas that we found useful are:

**Lemma** \text{maxTrA_nil}: \(\forall a \ l \ n \ V, ([], a) = \text{maxTrA} \ l \ n \ V \rightarrow l = [] \land V = []\).

**Lemma** \text{maxTrA_max}: \(\forall \text{max} \ l \ U b L x, (U, \text{max}) = \text{maxTrA} \ l \ b L \rightarrow (\text{In} \ x (b :: l) \rightarrow le \ x \ max)\).

**Lemma** \text{permutation_MxTrA}: \(\forall U \text{max} \ l \ b, (U, \text{max}) = \text{maxTrA} \ l \ b [] \rightarrow \text{permutation} (\text{max} :: U) (b :: l)\).

**Lemma** \text{maxTrA_n}: \(\forall \ l \ L1 \ U \text{max} a, (U,\text{max}) = \text{maxTrA} \ l \ a [] \leftrightarrow (L1 ++ U, \text{max}) = \text{maxTrA} \ l \ a \ L1\).

For proving the 'sorting' property, we have used the \text{induction} tactic for 17 times. There was no need to use the \text{functional induction} tactic.

5 Conclusions and Future Work

We have presented two different specifications and verification proofs for a tail recursive version of the Bubble-Sort algorithm, by using the \text{Theorema} and \text{Coq} systems. Below we mention the main differences between them.

In the \text{Theorema} specifications, the types are not explicitly declared, the functions can be partial and the recursive functions not terminating. In \text{Coq}, the specifications are typed, the functions are total and the recursive functions should terminate. In order to properly define induction schemas from the definition of recursive functions, we had to represent multiple arguments as one argument under the form of a tuple grouping them. The \text{Theorema} proofs use multisets and their properties defined ad-hoc by the user. The \text{Coq} proofs use lists and the permutation relation instead of multisets. In \text{Theorema} we have: 6 definitions, 2 algorithms, 2 theorems, 21 properties (from which 12 properties are specific to the certification of the algorithm). On the other side, in \text{Coq} we used about 30 lemmas.

The verification proofs in both systems required crucial human intervention, especially when performing the induction reasoning (e.g., finding the right induction variables and induction schemas), as well as automatic reasoning for executing specific tasks (e.g., the \text{lia} tactic for arithmetic reasoning in \text{Coq}). The proofs in \text{Theorema} are based on general inference rules, mainly for performing basic logical reasoning, as well as special inference rules, as those based on the natural properties of total order (e.g. transitivity). The general proof strategy was based on \text{cascading}, which requires the ad-hoc generation of new lemmas when the proof of the current goal fails. Most of the new lemmas had to be proved as the current goal, others have been imported from the standard libraries of the used system. In \text{Coq}, the definition for permutation presented in the paper and the related properties were user-defined, but could have also used other definitions, as those based on inductive predicates included in the \text{Sorting.Permutation} library.

The files in \text{Theorema} and the script in \text{Coq} described in Sections 3, 4 can be found at \text{https://members.loria.fr/SStratulat/files/LPAR2024.zip}.

As future work we consider to certify the \text{Bubble-Sort} with a flag (see [8]) in \text{Theorema} and \text{Coq}, and to increase the automation of proving and of finding necessary lemmata in \text{Theorema}. Another interesting research direction is to integrate the two systems in order to obtain natural style proofs that are also rigorously certified.
Acknowledgements

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

References


