Kalpa Publications in Computing

Volume 18, 2024, Pages 53–68

LPAR 2024 Complementary Volume

Certification of Tail Recursive Bubble–Sort in *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

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 rigurously 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

N. Bjørner, M. Heule and A. Voronkov (eds.), LPAR 2024C (Kalpa Publications in Computing, vol. 18), pp. 53–68

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 *Trimm* (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 *maxTrimm* that returns the maximum and the list without it. Then, the tail recursive sorting algorithm that uses the function *maxTrimm* 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 cerification of the tail recursive version together with the auxiliary function.

In *Theorema* the background theory, the algorithms, and the proof rules are composed adhoc 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 rigurously 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*[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

[1] www.wolfram.com/mathematica

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). $\langle\rangle$ is the empty list, and in the form head–tail this is denoted by $a \smile U$, (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 $\leq$). $\emptyset$ denotes the empty set, $\{\{a\}\}$ is the multiset containing the element a with multiplicity one, and $\mathcal{M}[U]$ denotes the multiset of the list U. $\uplus$ 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 ($el \leq U$ denotes that the element el is smaller or equal to each element of the list U, $U \leq el$ denotes that each member of the list U are smaller or equal to the element el); and between two lists ($U \leq V$ 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. $\underset{X}{\forall}$ "for all X", $\underset{X}{\exists}$ "exists X"), and Skolem constants have integer indices (e.g., U_0, a_0).

Basic definitions.

Definition 1. $$\underset{a,U}{\forall}\left(\begin{array}{c} IsSorted[\langle\rangle] \\ IsSorted[a \smile U] \Longleftrightarrow (a \leq U \wedge IsSorted[U]) \end{array}\right)$$

Definition 2. $$\underset{a,U}{\forall}\left(\begin{array}{c} \mathcal{M}[\langle\rangle] = \emptyset \\ \mathcal{M}[a \smile U] = \{\{a\}\} \uplus \mathcal{M}[U] \end{array}\right)$$

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.

$$\underset{a,b,U,V}{\forall}\left(\begin{array}{c} maxTrimm[a \smile U] = maxTrA[U, a, \langle\rangle] \\ maxTrA[\langle\rangle, a, V] = \langle V, a\rangle \\ maxTrA[b \smile U, a, V] = \begin{cases} maxTrA[U, b, V \frown a], & \textit{if } a \leq b \\ maxTrA[U, a, V \frown b], & \textit{if } b < a \end{cases} \end{array}\right)$$

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.

$$\underset{a,b,U,V}{\forall}\left(\begin{array}{c} BSort[a \smile U] = BSortA[maxTrA[U, a, \langle\rangle], \langle\rangle] \\ BSortA[\langle\langle\rangle, a\rangle, V] = a \smile V \\ BSortA[\langle b \smile U, a\rangle, V] = BSortA[maxTrA[U, b, \langle\rangle], a \smile V] \end{array}\right)$$

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 :

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

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

where the *count* function is defined as:

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

The *=?* notation represents the boolean equality that helps to compare two naturals.

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

```
Fixpoint maxTrA l a V :=
 match l with
   nil ⇒ (V, a)
| b :: U ⇒ if leb a b then maxTrA U b (V ++ [a]) else maxTrA U a (V ++ [b])
end.
```

```
Function BSortA p {wf (fun p1 p2 ⇒
                         Nat.lt (length (fst (fst p1))) (length ((fst (fst p2))))) p}:=
  match p with
   ((nil, a), V) ⇒ a :: V
  | (((b :: U), a), V) ⇒ BSortA ((maxTrA U b nil), (a :: V))
end.
```

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

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

Finally, *BSort* is defined as:

```
Definition BSort l :=
  match l with
    nil ⇒ nil
  | a :: U ⇒ BSortA ((maxTrA U a nil), nil)
end.
```

3 Verification in *Theorema*

For the verification of the sorting algorithm *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. $\underset{X}{\forall}\Big(\mathcal{M}[X] = \mathcal{M}[\mathit{BSort}[X]]\Big)$

Theorem 2. $\underset{X}{\forall}\Big(\mathit{IsSorted}[\mathit{BSort}[X]]\Big)$

3.1 Proof of Theorem 1

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

Property 1. $\underset{a,b,X}{\forall}\Big((a \smile X \leq b) \Longleftrightarrow (X \leq b \wedge a \leq b)\Big)$

Property 2. $\underset{a,b,X}{\forall}\Big((b \leq a \smile X) \Longleftrightarrow (b \leq X \wedge b \leq a)\Big)$

Property 3. $\underset{a,X}{\forall}\Big(\mathcal{M}[X \frown a] = \mathcal{M}[X] \uplus \{\{a\}\}\Big)$

Property 4. $\underset{X,Y}{\forall}\Big(\mathcal{M}[X \asymp Y] = \mathcal{M}[X] \uplus \mathcal{M}[Y]\Big)$

Property 5. $\underset{a,X}{\forall}\Big(\mathcal{M}[\langle X, a\rangle] = \mathcal{M}[X] \uplus \{\{a\}\}\Big)$

Property 6. $\underset{X,Y}{\forall}\Big(\mathcal{M}[\langle X, Y\rangle] = \mathcal{M}[X] \uplus \mathcal{M}[Y]\Big)$

Property 7. $\underset{a,U,V}{\forall}\Big(\mathcal{M}[\mathit{maxTrA}[U, a, V]] = \mathcal{M}[U] \uplus \{\{a\}\} \uplus \mathcal{M}[V]\Big)$

Property 8. $\underset{U,V}{\forall}\Big(\mathcal{M}[\mathit{BSortA}[U, V]] = \mathcal{M}[U] \uplus \mathcal{M}[V]\Big)$

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

$$\mathcal{M}[a_0 \smile U_0] = \mathcal{M}[\mathit{Bubble\text{-}Sort}[a_0 \smile U_0]] \tag{1}$$

which becomes

$$\mathcal{M}[a_0 \smile U_0] = \mathcal{M}[\mathit{BSortA}[\mathit{maxTrA}[U_0, a_0, \langle\rangle], \langle\rangle]] \tag{2}$$

By Definition 2 the goal becomes:

$$\{\{a_0\}\} \uplus \mathcal{M}[U_0] = \mathcal{M}[\mathit{BSortA}[\mathit{maxTrA}[U_0, a_0, \langle\rangle], \langle\rangle]] \tag{3}$$

By Property 8 the goal becomes:

$$\{\{a_0\}\} \uplus \mathcal{M}[U_0] = \mathcal{M}[\mathit{maxTrA}[U_0, a_0, \langle\rangle]] \uplus \mathcal{M}[\langle\rangle] \tag{4}$$

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

$$\{\{a_0\}\} \uplus \mathcal{M}[U_0] = \mathcal{M}[\mathit{maxTrA}[U_0, a_0, \langle\rangle]] \tag{5}$$

By Property 7 the goal becomes:

$$\{\{a_0\}\} \uplus \mathcal{M}[U_0] = \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[\langle\rangle] \tag{6}$$

This holds by Definition 2 and by properties of multiset union. $\square$

The proof of Property 7.

Proof. Prove $\underset{a,U,V}{\forall}\Big(\mathcal{M}[maxTrA[U,a,V]] = \mathcal{M}[U] \uplus \{\{a\}\} \uplus \mathcal{M}[V]\Big)$
by induction on U:
Base case: Take a_0, V_0 arbitrary but fixed and prove

$$\mathcal{M}[maxTrA[\langle\rangle, a_0, V_0]] = \mathcal{M}[\langle\rangle] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V] \tag{7}$$

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

$$\mathcal{M}[maxTrA[\langle\rangle, a_0, V_0]] = \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{8}$$

By Algorithm 1 the goal becomes:

$$\mathcal{M}[\langle V_0, a_0\rangle] = \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{9}$$

By Property 5 the goal becomes:

$$\mathcal{M}[V_0] \uplus \{\{a_0\}\} = \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{10}$$

This holds by the commutativity of $\uplus$.
Inductive step: Take b_0, U_0 arbitrary but fixed (a, V remains universally quantified), assume:

$$\underset{a,V}{\forall}(\mathcal{M}[maxTrA[U_0, a, V]] = \mathcal{M}[U_0] \uplus \{\{a\}\} \uplus \mathcal{M}[V]) \tag{11}$$

and prove

$$\underset{a,V}{\forall}(\mathcal{M}[maxTrA[b_0 \smile U_0, a, V]] = \mathcal{M}[b_0 \smile U_0] \uplus \{\{a\}\} \uplus \mathcal{M}[V]) \tag{12}$$

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

$$\mathcal{M}[maxTrA[b_0 \smile U_0, a_0, V_0]] = \{\{b_0\}\} \uplus \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{13}$$

We prove (13) by cases using Algorithm 1:
Case 1: $a_0 \leq b_0$. The goal becomes:

$$\mathcal{M}[maxTrA[U_0, b_0, V_0 \frown a_0]] = \{\{b_0\}\} \uplus \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{14}$$

By induction hypothesis (11) the goal becomes:

$$\mathcal{M}[U_0] \uplus \{\{b_0\}\} \uplus \mathcal{M}[V_0 \frown a_0] = \{\{b_0\}\} \uplus \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{15}$$

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 *BSortA* always consists of a pair between a list and an element. Therefore it is sufficient to show:

$$\underset{a,U,V}{\forall}\Big(\mathcal{M}[BSortA[\langle U, a\rangle, V]] = \mathcal{M}[U] \uplus \{\{a\}\} \uplus \mathcal{M}[V]\Big) \tag{16}$$

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*:

$$\underset{U}{\forall}\Big(\mathcal{M}[BSortA[\langle U, a_0\rangle, V_0]] = \mathcal{M}[U] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0]\Big) \tag{17}$$

Base case: Prove

$$\mathcal{M}[BSortA[\langle\langle\rangle, a_0\rangle, V_0]] = \mathcal{M}[\langle\rangle] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{18}$$

By Definition 2 and by property unit the goal becomes:

$$\mathcal{M}[BSortA[\langle\langle\rangle, a_0\rangle, V_0]] = \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{19}$$

By Algorithm 2 the goal becomes:

$$\mathcal{M}[a_0 \smile V_0] = \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{20}$$

By Definition 2 the goal becomes:

$$\{\{a_0\}\} \uplus \mathcal{M}[V_0] = \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{21}$$

This holds by reflexivity of equality.

Induction step: Take b_0, U_0 arbitrary, but fixed and prove

$$\mathcal{M}[BSortA[\langle b_0 \smile U_0, a_0\rangle, V_0]] = \mathcal{M}[b_0 \smile U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{22}$$

By Definition 2 the goal becomes:

$$\mathcal{M}[BSortA[\langle b_0 \smile U_0, a_0\rangle, V_0]] = \{\{b_0\}\} \uplus \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{23}$$

By Algorithm 2 the goal becomes:

$$\mathcal{M}[BSortA[maxTrA[U_0, b_0, \langle\rangle], a_0 \smile V_0] = \{\{b_0\}\} \uplus \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{24}$$

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:

$$\mathcal{M}[maxTrA[U_0, b_0, \langle\rangle]] \uplus \mathcal{M}[a_0 \smile V_0] = \{\{b_0\}\} \uplus \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{25}$$

By Property 7 the goal becomes:

$$\mathcal{M}[U_0] \uplus \{\{b_0\}\} \uplus \mathcal{M}[\langle\rangle] \uplus \mathcal{M}[a_0 \smile V_0] = \{\{b_0\}\} \uplus \mathcal{M}[U_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[V_0] \tag{26}$$

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

□

3.2 Proof of Theorem 2

For proving $\underset{X}{\forall}\Big(IsSorted[BSort[X]]\Big)$ 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.

$$\left(\begin{array}{l}\underset{G,H}{\forall}\Big(E[G,H] \Longleftrightarrow \big(\underset{a,b,U,V}{\exists}(G = \langle b \smile U, a, V\rangle \wedge ((a \leq b) \wedge (H = \langle U, b, V \frown a\rangle))\vee \\ \qquad\qquad\qquad (b < a \wedge H = \langle U, a, V \frown b\rangle)))\big)\Big) \\ \underset{G,H}{\forall}\big(E^*[G,H] \Longleftrightarrow (G = H \vee (\underset{K}{\exists}(E^*[G,K] \wedge E[K,H]))))\big)\end{array}\right)$$

Definition 4.

$$\underset{G}{\forall}\Big(P[G] \Longleftrightarrow \big(\underset{a,U,V}{\exists}(G = \langle U, a, V\rangle \wedge V \leq a)\big)\Big)$$

The following is an elementary consequence of this definition:

Property 9. $\underset{a,U,V}{\forall}\Big(P[\langle U, a, V\rangle] \Longrightarrow V \leq a\Big)$

We also need the following properties in the knowledge base:

Property 10. $\underset{a,b,L}{\forall}\Big(b \leq a \wedge L \leq a \Longleftrightarrow b \smile L \leq a\Big)$

Property 11. $\underset{a,b,L}{\forall}\Big(a \leq b \wedge V \leq a \Longleftrightarrow V \frown a \leq b\Big)$

Property 12. $\underset{x,y\neq\langle\rangle,z}{\forall}\Big(x \leq y \wedge y \leq z \Longrightarrow x \leq z\Big)$

Property 13. $\underset{G,H}{\forall}\Big(E[G,H] \Longrightarrow (P[G] \Longrightarrow P[H])\Big)$

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:

$$G = \langle b_0 \smile U_0, a_0, V_0\rangle \tag{27}$$

and

$$(a_0 \leq b_0 \wedge (H = \langle U_0, b_0, V_0 \frown a_0\rangle)) \vee (b_0 < a_0 \wedge H = \langle U_0, a_0, V_0 \frown b_0\rangle) \tag{28}$$

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

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

$$P\left[\langle b_0 \smile U_0, a_0, V_0\rangle\right] \tag{29}$$

From this by (9) we obtain

$$V_0 \leq a_0 \tag{30}$$

In order to prove $P[H]$, by the definition of P, we prove:

$$\underset{a,U,V}{\exists}(H = \langle U, a, V\rangle \wedge V \leq a) \tag{31}$$

We take $U = U_0$ and prove:

$$\underset{a,V}{\exists}\left(H = \langle U_0, a\rangle \wedge V \leq a\right)$$

by cases using the disjunction (28).

Case 1: $a_0 \leq b_0,\ H = \langle U_0, b_0, V_0 \frown a_0\rangle$

We take $a = b_0$, V=$V_0 \frown a_0$ and from 30 and $a_0 \leq b_0$ by properties of ordering we obtain: $V_0 \frown a_0 \leq b_0$ that is necessary for $P[\langle U_0, b_0, V_0 \frown a_0\rangle]$.

Case 2: $b_0 < a_0,\ \ H = \langle U_0, a_0, V_0 \frown b_0\rangle$

We take $a = a_0$, V=$V_0 \frown b_0$ and from 30 and $b_0 < a_0$ by properties of ordering we obtain: $V_0 \frown b_0 \leq a_0$ that is necessary for $P[\langle U_0, a_0, V_0 \frown b_0\rangle]]$ □

Property 14. $\underset{G,H}{\forall}\Big(E^*[G,H] \Longrightarrow (P[G] \Longrightarrow P[H])\Big)$

The proof of Property 14 is straightforward by Definion 3, and by Property 13.

Property 15. $\underset{a,b,U,V,W}{\forall}\Big(E^*[\langle U, a, \langle\rangle\rangle, \langle W, b, V\rangle] \Longrightarrow V \leq b\Big)$

The proof of Property 15 uses Definion 4, and the property $a \leq \langle\rangle$.

The algorithm *maxTrA* terminates because the only recursive calls from Algorithm 1 reduce the argument $b \smile U$ 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. $\underset{b,U}{\forall}\underset{a,V}{\exists}\Big(maxTrA[U, b, \langle\rangle] = \langle V, a\rangle] \wedge V \leq a\Big)$

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

Definition 5.

$$\left(\begin{array}{l}\underset{G,H}{\forall}\Big(Es[G,H] \Longleftrightarrow \big(\underset{a,b,U,V}{\exists}(G = \langle\langle b \smile U, a\rangle, V\rangle \wedge H = \langle maxTrA[U, b, \langle\rangle], a \smile V\rangle)\big)\Big)\\ \underset{G,H}{\forall}\big(Es^*[G,H] \Longleftrightarrow \big(G = H \vee \underset{K}{\exists}(Es^*[G,K] \wedge Es^*[K,H])\big)\big)\end{array}\right)$$

Definition 6. $\underset{G}{\forall}\Big(Ps[G] \Longleftrightarrow \big(\underset{a,U,V}{\exists}(G = \langle\langle U, a\rangle, V\rangle \wedge IsSorted[a \smile V] \wedge U \leq V)\big)\Big)$

The following is an elementary consequence of this definition:

Property 17. $\underset{a,U,V}{\forall}\Big(Ps[\langle\langle U, a\rangle, V\rangle] \Longrightarrow (IsSorted[a \smile V] \wedge U \leq a \smile V)\Big)$

We prove now that the evolution relation transports the property.

Property 18. $\underset{G,H}{\forall}\Big(Es[G,H] \Longrightarrow (Ps[G] \Longrightarrow Ps[H])\Big)$

Proof. The proof of Property 18 applies general inference rules and by Algorithm 2 the goal is

$$\underset{a,b,U,V}{\exists}\Big(G_0 = \langle\langle b \smile U, a\rangle, V\rangle \wedge H_0 = \langle maxTrA[U, b, \langle\rangle], a \smile V\rangle\Big) \tag{32}$$

By the above we can find a_0, b_0, U_0, V_0 such that

$$G_0 = \langle\langle b_0 \smile U, a_0\rangle, V_0\rangle \tag{33}$$

and

$$H_0 = \langle maxTrA[U_0, b_0, \langle\rangle], a \smile V_0\rangle \tag{34}$$

From the previous assumption by (33) we obtain

$$Ps[\langle\langle b_0 \smile U_0, a_0\rangle, V_0\rangle] \tag{35}$$

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

$$\underset{a,U,V}{\exists}\Big(H_0 = \langle\langle U, a\rangle, V\rangle \wedge IsSorted[a \smile V] \wedge U \leq a \smile V\Big) \tag{36}$$

We instantiate Property 16 with $a \to b_0$ and $U \to U_0$ and take V_1, b_1 such as:

$$maxTrA[U_0, b_0, \langle\rangle] = \langle V_1, b_1\rangle \tag{37}$$

and

$$V_1 \leq b_1 \tag{38}$$

By applying Definition 2, Properties 7, 5, and multisets preserve ordering we obtain: $b_1 \leq a_0 \smile V_0$, and $V_1 \leq a_0 \smile V_0$. From (34) by using the derived assumptions we obtain

$$H_0 = \langle\langle V_1, b_1\rangle, a_0 \smile V_0\rangle \wedge IsSorted[b_1 \smile (a_0 \smile V_0)] \wedge V_1 \leq b_1 \smile (a_0 \smile V_0) \tag{39}$$

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

$$IsSorted[b_1 \smile (a_0 \smile V_0)] \wedge V_1 \leq b_1 \smile (a_0 \smile V_0) \tag{40}$$

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

□

Property 19. $\underset{G,H}{\forall}\Big(Es^*[G, H] \Longrightarrow (Ps[G] \Longrightarrow Ps[H])\Big)$

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

Property 20. $\underset{a,b,U,V}{\forall}\Big(Es^*[\langle maxTrA[U, a, \langle\rangle], \langle\rangle\rangle, \langle\langle\rangle, b\rangle, V\rangle] \Longrightarrow IsSorted[b \smile V]])\Big)$

Proof. Take a_0, b_0, U_0, V_0 arbitrary but fixed. Assume

$$Es^*[\langle maxTrA[U_0, a_0, \langle\rangle], \langle\rangle\rangle, \langle\langle\rangle, b_0\rangle, V_0\rangle]\big) \tag{41}$$

and prove

$$IsSorted[b_0 \smile V_0] \tag{42}$$

(41) by Property 19 becomes:

$$Ps[\langle maxTrA[U_0, a_0, \langle\rangle], \langle\rangle\rangle] \Longrightarrow Ps[\langle\langle\langle\rangle, b_0\rangle, V_0\rangle]\big) \tag{43}$$

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

$$Ps[\langle\langle\langle\rangle, b_0\rangle, V_0\rangle]\big) \tag{44}$$

From (44) by Definition 6 we obtain $IsSorted[b_0 \smile V_0] \wedge \langle\rangle \leq b_0 \smile V_0$ which proves our goal. □

The algorithm *BSortA* terminates because the only recursive call of the Algorithms 2 reduces the argument $\langle b \smile U, a\rangle$ to $maxTrA[U, b, \langle\rangle]$. This follows from Property 7 and to properties of multisets:

$\mathcal{M}[maxTrA[U, b, \langle\rangle]] = \mathcal{M}[U] \uplus \{\{b\}\}$ is strictly included in $\mathcal{M}[U] \uplus \{\{a, b\}\} = \mathcal{M}[\langle b \smile U, a\rangle]$

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 $a \smile V$ which by Property 20 has the property $IsSorted[b \smile V]$.

From this follows easily:

Property 21. $\underset{a,U}{\forall}\ \underset{b,V}{\exists}\Big(BSortA[maxTrA[U, a, \langle\rangle], \langle\rangle] = b \smile V \wedge IsSorted[b \smile V]])\Big)$

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 : $is\_a\_sorting\_algorithm\ BSort.$

where $is\_a\_sorting\_algorithm$ is the function used to check the soundness property to be satisfied by the sorting function f given as argument:

`Definition` $is\_a\_sorting\_algorithm\ (f\colon\ list\ nat \to list\ nat) :=$
$\forall\ al,\ permutation\ (f\ \ al)\ al \wedge IsSorted\ (f\ \ al).$

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: $\forall\ x\ l,\ In\ x\ l \leftrightarrow In\ x\ (BSort\ \ l).$

`Lemma` $BSort\_count$: $\forall\ x\ l,\ count\ x\ l = count\ x\ (BSort\ \ l).$

The proof of the BS_in_equiv lemma is based on the following lemmas:

`Lemma` $BSortA\_in\_rev$: $\forall\ x\ p,\ In\ x\ (BSortA\ p) \to (snd\ (fst\ p)) = x \vee In\ x\ (fst\ (fst\ p)) \vee In\ x\ (snd\ p).$

`Lemma` BS_in_rev : $\forall\ x\ l,\ In\ x\ (BSort\ \ l) \to In\ x\ l.$

`Lemma` $maxTrA\_in\_fst$: $\forall\ U\ a\ b\ L,\ In\ b\ L \to In\ b\ (fst\ (maxTrA\ \ U\ \ a\ \ L)).$

`Lemma` $maxTrA\_in$: $\forall\ U\ x\ b\ L,\ In\ x\ (U{+}{+}L{+}{+}[b]) \to snd\ (maxTrA\ \ U\ \ b\ \ L) = x \vee In\ x\ (fst\ (maxTrA\ \ U\ \ b\ \ L)).$

`Lemma` $BSortA\_in$: $\forall\ x\ p,\ (snd\ (fst\ p)) = x \vee In\ x\ (fst\ (fst\ p)) \vee In\ x\ (snd\ p) \to In\ x\ (BSortA\ p).$

`Lemma` BS_in : $\forall\ x\ l,\ In\ x\ l \to In\ x\ (BSort\ \ l).$

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

`Lemma` $count\_app$: $\forall\ x\ l1\ \ l2,\ count\ x\ (app\ l1\ \ l2) = count\ x\ l1\ + (count\ x\ l2).$

`Lemma` $count\_maxTrA$: $\forall\ x\ U\ b\ L,\ count\ x\ (U{+}{+}L{+}{+}[b]) =$
$(\texttt{if}\ x =?\ snd\ (maxTrA\ \ U\ \ b\ \ L)\ \texttt{then}\ 1\ \texttt{else}\ 0) + count\ x\ (fst\ (maxTrA\ \ U\ \ b\ \ L)).$

`Lemma` $BSortA\_count : \forall\ x\ p,\ count\ x\ (BSortA\ p) =$
$((\texttt{if}\ x =?\ (snd\ (fst\ p))\ \texttt{then}\ 1\ \texttt{else}\ 0) + count\ x\ (snd\ p)) + (count\ x\ (fst\ (fst\ p)))$.

`Lemma` $BSortA\_count : \forall\ x\ p,\ count\ x\ (BSortA\ p) =$
$((\texttt{if}\ x =?\ (snd\ (fst\ p))\ \texttt{then}\ 1\ \texttt{else}\ 0) + count\ x\ (snd\ p)) + (count\ x\ (fst\ (fst\ p)))$.

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` $maxTrN\_maxTrA : \forall\ l\ a\ U,\ maxTrA\ l\ a\ U = (U \mathbin{++} fst\ (maxTrN\ l\ a),$
$snd\ (maxTrN\ l\ a))$.

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 ($maxTrA\ l\ n1$ []) by ($maxTrN\ l\ n1$). Finally, in the proof of the 'sorting' property, we have replaced *maxTrA* by *maxTrN* using the *maxTrN_maxTrA_nil* lemma before calling *BSort_is_sorted*.

Other lemmas that we found useful are:

`Lemma` *maxTrA_nil* : $\forall\ a\ l\ n\ V$, ([], a) = $maxTrA\ l\ n\ V \rightarrow l$ = [] $\wedge\ V$ = [].

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

`Lemma` *permutation_MaxTrA*: $\forall\ U\ max\ l\ b$, $(U, max) = maxTrA\ l\ b$ [] $\rightarrow permutation\ (max :: U)\ (b :: l)$.

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

For proving the 'sorting' property, we have used the *induction* tactic for 17 times. There was no need to use the *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 *Theorema* and Coq systems. Below we mention the main differences between them.

In the *Theorema* specifications, the types are not explicitly declared, the functions can be partial and the recursive functions not terminating. In 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 *Theorema* proofs use multisets and their properties defined ad-hoc by the user. The Coq proofs use lists and the permutation relation instead of multisets. In *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 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 *lia* tactic for arithmetic reasoning in Coq). The proofs in *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 *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 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 *Sorting.Permutation* library.

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

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

[1] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science*. Springer, 2004.

[2] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *IFM 2017*, pages 102–110. Springer International Publishing, 2017.

[3] J. Bockenek, P. Lammich, Y. Nemouchi, and B. Wolff. Using Isabelle/UTP for the verification of sorting algorithms: A case study. EasyChair Preprint no. 944, 2019.

[4] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey on the Theorema project. In *In International Symposium on Symbolic and Algebraic Computation*, pages 384–391. ACM Press, 1997.

[5] B. Buchberger, T. Jebelean, T. Kutsia, A. Maletzky, and W. Windsteiger. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *Journal of Formalized Reasoning*, 9(1):149–185, 2016.

[6] M. P. F. Burgos. Formalization of sorting algorithms in Isabelle/HOL. Master's thesis, Vrije Universiteit Amsterdam, 2019.

[7] I. Dramnesc and T. Jebelean. Synthesis of List Algorithms by Mechanical Proving. *Journal of Symbolic Computation*, 68:61–92, 2015.

[8] I. Dramnesc and T. Jebelean. Deductive Synthesis of Bubble-Sort Using Multisets. In *SAMI 2020*, pages 165–172. IEEE, 2020.

[9] I. Dramnesc and T. Jebelean. Deductive synthesis of Min-Max-Sort using multisets. In *SACI 2020*, pages 165–172. IEEE, 2020.

[10] I. Dramnesc and T. Jebelean. Synthesis of sorting algorithms using multisets in Theorema. *Journal of Logical and Algebraic Methods in Programming*, 119(100635), 2020.

[11] I. Dramnesc and T. Jebelean. Mechanical Verification of Insert-Sort and Merge-Sort Using Multisets in Theorema. In *SISY 2023*, pages 55–60. IEEE, 2023.

[12] I. Dramnesc, T. Jebelean, and S. Stratulat. Combinatorial Techniques for Proof-based Synthesis of Sorting Algorithms. In *SYNASC 2015*, pages 137–144, 2015.

[13] I. Dramnesc, T. Jebelean, and S. Stratulat. Certification of Sorting Algorithms Using Theorema and Coq. In *SCSS 2024*. Submitted, 2024.

[14] J. C. Filliâtre. Deductive Program Verification with Why3 a tutorial (2013).

[15] J. C. Filliâtre and A. Paskevich. Why3 — Where Programs Meet Provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] Jean-Christophe Filliâtre and Nicolas Magaud. Certification of Sorting Algorithms in the Coq System. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[17] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism,*, pages 479–490, 1980.

[18] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison Wesley, 2 edition, 1998.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[20] D. Petrovic. Verification of selection and heap sort using locales. *Arch. Formal Proofs*, 2014, 2014.

[21] S. Quarfot Orrevall and A. Gengelbach. *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, 2020-11-04T08:12:02.574+01:00 2020.

[22] M. Safari and M. Huisman. A Generic Approach to the Verification of the Permutation Property of Sequential and Parallel Swap-Based Sorting Algorithms. In Brijesh Dongol and Elena Troubitsyna, editors, *Proceedings of IFM 2020*, volume 12546 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2020.

[23] The Coq development team. *The Coq Reference Manual*. INRIA, 2020. http://coq.inria.fr/doc.

[24] E. Tushkanova, A. Giorgetti, and O. Kouchnarenko. Specifying and Proving a Sorting Algorithm. Technical report, Laboratoire d'Informatique de l'Université de Franche-Comte, 2009.

[25] W. Windsteiger. Theorema 2.0: A system for mathematical theory exploration. In *ICMS'2014*, volume 8592 of *LNCS*, pages 49–52, 2014.

[26] Y. Zhang, Y. Zhao, and D. Sanán. A verified Timsort C implementation in Isabelle/HOL. *CoRR*, abs/1812.03318, 2018.