Proving the Correctness of the Merge–Sort Algorithm with *Theorema*

Adrian Craciun* and Bruno Buchberger
Research Institute for Symbolic Computation,
4232 Hagenberg, Austria
$\{acraciun, buchberg\}\ @risc.uni - linz.ac.at$

Abstract. This paper presents the proof of the correctness of the merge–sort algorithm, using the *Theorema* system. The problem is specified in the language provided by the system, having as underlying theory the theory of tuples. Tuples are represented in terms of sequences. The proof techniques for sequences, and their implementation in *Theorema*, in the form of the sequence provers are presented. The correctness theorem is proved by doing a top down exploration. The exploration paradigm which introduced here could be called "lazy thinking paradigm": deliberately don't provide any knowledge in terms of intermediate lemmata, until failing proofs of the correctness theorem (and its subtheorems) give a natural suggestion which intermediate lemmata could be useful for constructing a successful proof. The suggested intermediate lemmata are then handled by the same "lazy thinking paradigm" in a recursive way. Several exploration cycles are needed to complete the proof.

AMS Subject Classification: 68Q40

Keywords and phrases: automatic algorithm verification, sequence variables, theory exploration, lazy thinking paradigm.

1 Introduction

The correctness of algorithms is an important issue in computer science. Automated support for proving algorithms correct would prove an excellent tool for algorithm developers. This paper addresses this issue.

Theorema (see [6]) is a system that aims at providing automated support for mathematical activities - proving, solving, computing, theory exploration. The language provided by the system is a version of higher order predicate logic, which

*Work supported by the Calculemus European Project and by the Austrian Science Foundation (FWF) under project SFB F1302

should make it attractive to use for persons with some formal training in mathematics. This also makes it a useful tool in the training of mathematics and computer science students. A big part of the algorithms presented in introductory computer science courses can be formulated in terms of lists (tuples). One such algorithm is the merge–sort algorithm. The algorithm is specified in *Theorema*, and it is proved correct using the proving capabilities provided by the system.

Section 2 presents briefly the underlying theory of tuples, how these can be represented in terms of sequences and the formulation of the algorithm and of its correctness theorem in the language of *Theorema*. **Section 3** contains the presentation of the proof rules for sequences, their implementation in the frame of the *Theorema* proving mechanism, and their usage. The exploration cycle that leads to the proof of the correctness of merge–sort is described in **Section 4**. The **Appendix** contains the definitions relevant for the proofs presented in **Section 4**.

In the following, special fonts and font sizes are used to reprezent *Theorema* `input` (typewriter fonts), output of the system (proof text or other information-small fonts). In the proof fragments presented, [...] represents a part of the proof that is irrelevant to this presentation. The "lazy thinking paradigm" was introduced in a preliminary technical report by the second author. The details of the proofs and the implementation of the proof techniques for sequences in the *Theorema* system, continuing earlier work on sequence induction (see [5], [11]), were worked out by the first author.

2 The Merge–Sort Algorithm. Specification in *Theorema*

2.1 The Theory of Tuples

Tuples (nested lists) are objects constructed from an arbitrary finite number of objects in a specific order. In *Theorema*, tuples are represented with angle brackets. Examples:

$\langle 1, 2, 3\rangle$

$\langle 1, \langle a, a\rangle, d, \langle b, c\rangle\rangle$.

The question is how to represent tuples, in order to develop the theory about them. However, independent of the representation, some notions that describe operations on tuples are needed. The tuple theory must contain:

- a tuple constructor: a function that takes an arbitrary finite number of objects into a tuple;

- *is-tuple*: a predicate that tests whether a given object is a tuple. This can be used also as a "type", referring to objects as tuples, without representing them explicitly;

- basic tuple operations: concatenation of two tuples, prepending and appending an element to a tuple, reversing a tuple, selectors (first element, last element) etc;

- other operations: element of tuple, the length of a tuple, deletion of an element from a tuple, insertion of an element in a tuple;

- if there is an ordering on the elements in tuples: (upper, lower) bound of a tuple, sorting of elements in tuples.

There are several ways to represent tuples. One way would be to see them as indexed lists, with the elements retrieved according to their index. One other possibility is (like in this in this presentation), is to use *sequences* .

2.2 Representing Tuples in Terms of Sequences

Sequence variables (see [2], [7]) are variables that can be instantiated with an arbitrary finite number of terms. In *Theorema* they are denoted by overbarred identifiers, $\overline{\mathrm{x}}$, $\overline{\mathrm{y}}$ ...
For example, if $\overline{\mathrm{x}}$ is a sequence variable, then

$\overline{\mathrm{x}} \leftarrow$ (substituting the empty sequence for the sequence variable)

$\overline{\mathrm{x}} \leftarrow$ t, u, v (substituting a sequence of terms for the sequence variables)

$\overline{\mathrm{x}} \leftarrow$ t, $\overline{\mathrm{t}}$ (substituting a sequence of terms that contains also a sequence, for the sequence variable).

From the informal description, the following are tuples:

$\langle\overline{\mathrm{x}}\rangle$

$\langle \mathrm{a, b}, \overline{\mathrm{c}}\rangle$

Tuples can expressed in terms of sequence variables in the following way:

Definition["is tuple", any[X],
  is-tuple[X] $\Leftrightarrow \underset{\overline{x}}{\exists}(X = \langle\overline{x}\rangle)$]

2.3 The Merge–Sort Algorithm

The merge–sort algorithm is an operation on tuples that takes a tuple and returns its sorted permuted version. It is of the type divide-and-conquer (see [1]), which suggests to define the solution function, S, for any X as:

$$\mathtt{S[X]=} \begin{array}{lll} special[X] & \Leftarrow & is-special[X] \\ compose[S[left-part[X]], S[right-part[X]]] & \Leftarrow & otherwise \end{array}$$

For the algorithm to terminate, *left-part* and *right-part* must satisfy

¬is–special[X]⇒is–greater[X,left–part[X]]

¬is–special[X]⇒is–greater[X,right–part[X]],

where is–greater is a Noetherian partial ordering (for which Noetherian induction is possible) on the objects X which are of interest. In the case of tuples, a possible Noetherian ordering is the "is-longer-tuple" relation.

Hence the next more concrete version of the merge–sort algorithm for tuples is:

Algorithm["merge-sort", any[is-tuple[A]],

$$\mathtt{stmg[A]:=} \begin{array}{lll} A & \Leftarrow & |A|\leq 1 \\ mg[stmg[lsp[A]],\ stmg[rsp[A]]] & \Leftarrow & otherwise \end{array} \quad \mathtt{]}$$

where

stmg- sort by merging, an unary function with tuples as arguments,

mg - merge, a binary function that takes two tuples and returns a new tuple,

lsp, *rsp* - left split and right split of a tuple. unary functions that takes argument a tuple and returns another tuple,

$|\ldots|$ - an unary function that takes a tuple as argument and returns its length.

The empty tuple and the tuple with just one element are considered special tuples. These are sorted. For the other case, the tuple is split in two parts and the algorithm applied recursively to each of the splits, then the results are merged.

The aim is to prove that the algorithm is correct. This means that the result must be a tuple, which is a permuted version of the input, and it is sorted:

```
Proposition["correctness of merge-sort", any[is-tuple[A]],
    is-tuple[stmg[A]]
    stmg[A]≈ A
    is − sorted[stmg[A]]],
```

where

$\approx$ - "*is-permuted-version*", binary predicate on tuples,

is-sorted - binary predicate on tuples.

The chosen representation for tuples is suitable for the way the merge−sort algorithm was defined (i.e. recursively). For proving the correctnes the available sequence provers will be used. These will be presented in the next section, along with the proof techniques for sequence.

3 Sequence Provers in *Theorema*

3.1 *Theorema* Terminology

A *proof situation* $\langle \mathbf{K}, \mathbf{G} \rangle$ is made up from a *knowledge base* **K**, and a *proof goal* **G**, and read it "Prove **G** from **K**". The proof goal is typically one formula, whereas the knowledge base contains a collection of formulae.

Special provers execute individual *proof steps*, that reduce the proof situation (to a simpler one). The reduction is done according to *inference rules* that are implemented in the special prover.

The *user provers* represent the mechanism for combining several special provers. It implements the proof strategies, that guide the reduction of the proof situation to the final proof step, that contains *True* in the goal (and say that the initial goal was proved).

Proof objects in *Theorema* are trees of proof situations. The nodes contain a list of subgoals that have to be proved. There are *AND nodes* (all the subgoals have to be proved for the proof to succeed) and *OR nodes* (it suffices that one of the subgoals are proved for the proof to succeed). For details, see [6].

3.2 Proof Techniques for Sequences

In the following the proof rules for sequences are presented, in the form of the sequent notation (see [8]). The goal is a universally quantified formula, where (at least one of) the bounded variables are sequence variables. Every application of one of the rules has as effect the elimination of an universal quantifier from the formula. The proof rules for sequences only work on the goal, so the rules show the transformation of the goal.

- *arbitrary but fixed (or explicit representation):*

$$\frac{\overline{x}_0\text{arb., fixed}: \quad \Phi_{\overline{x}\leftarrow\overline{x_0}}}{\underset{\overline{x}}{\forall}\Phi}$$

- *case distinction:*

$$\frac{\Phi_{\overline{x}\leftarrow} \qquad\qquad x_0,\ \overline{x}_0\text{arb., fixed}: \quad \Phi_{\overline{x}\leftarrow\ x_0,\overline{x_0}}}{\underset{\overline{x}}{\forall}\Phi}$$

$$\frac{\Phi_{\overline{x}\leftarrow} \qquad\qquad x_0,\ \overline{x}_0\text{arb., fixed}: \quad \Phi_{\overline{x}\leftarrow\overline{x_0},\ x_0}}{\underset{\overline{x}}{\forall}\Phi}$$

Look at two cases, the base case, when we substitute the empty sequence for the sequence variable to be eliminated, and the general case, when the sequence variable to be eliminated is substituted with an unempty arbitrary sequence ($x_0,\ \overline{x_0}$ is a left unempty sequence, $\overline{x_0},\ x_0$ is a right unempty sequence). Other case distinction rules can be imagined, e.g. when the sequence variable to be eliminated is sunstituted with a sequence that has (at least) an inside unempty element ($\overline{x_0}$, x_0, $\overline{y_0}$).

- *structural induction* (introduced in [5], [11]):

$$\frac{\Phi_{\overline{x}\leftarrow} \qquad\qquad x_0,\ \overline{x}_0\text{arb., fixed}: \ \Phi_{\overline{x}\leftarrow\ \overline{x_0}} \quad\Rightarrow\quad \Phi_{\overline{x}\leftarrow\ x_0,\overline{x_0}}}{\underset{\overline{x}}{\forall}\Phi}$$

$$\frac{\Phi_{\overline{x}\leftarrow} \qquad\qquad x_0,\ \overline{x}_0\text{arb., fixed}: \ \Phi_{\overline{x}\leftarrow\ \overline{x_0}} \quad\Rightarrow\quad \Phi_{\overline{x}\leftarrow\overline{x_0},\ x_0}}{\underset{\overline{x}}{\forall}\Phi}$$

For structural induction, there are again two cases, the base case, where the empty sequence is substituted for the sequence variable to be eliminated, and the induction step, where the formula is assumed true for an arbitrary sequence substituted for the sequence variable to be eliminated, in the induction hypothesis, and it is proved for a longer (with one element) sequence. Again, this element can be added at the beginning or at the end of the sequence, thus obtaining two proof rules.

Case distinction is in fact a special case of the structural induction, where the induction hypothesis is not needed in the proof.

- *well founded induction:*

$$\frac{\overline{x_0}\text{ arb., fixed : } \left(\underset{\overline{y}}{\forall}(\langle\overline{x_0}\rangle \succ \langle\overline{y}\rangle \Rightarrow \Phi_{\overline{x}\leftarrow\overline{y}})\right) \;\Rightarrow\; \Phi_{\overline{x}\leftarrow\overline{x_0}}}{\underset{\overline{x}}{\forall}\Phi}$$

In the case of the well-founded induction, the well−founded ordering ($\succ$) is passed as an argument by the user. Its properties (well−foundedness) have to be established in the theory. The proof rule described above is dependent on the theory of tuples - the ordering that appears in the formula is an ordering on tuples. For other theories, different proof rules must be specified. The other proof rules are general, so they can be used in any theory with sequences.

3.3 Sqns - The Sequence Special Prover

The system provides information about the sequence special prover. The following are commands executed in *Theorema*:

`?Sqns`

Sqns[goal, kb, af] represents the implementation of several proof techniques for sequences (and tuples): arbitrary but fixed, case distinction, induction.

goal - a universally quantified formula that contains sequence variables under the quantifier
kb - the knowledge base
af - additional facts,used in the proof process

The user can influence the behaviour of the prover by specifying options for it. The options of the sequence prover are given by:

`Options[Sqns]`

{*SqnsWFOrdering*→None,SqnsProofTechniques→{*ABF*, *CDLeft*, *IndLeft*}}

And details for each option:

`?SqnsWFOrdering`

Option of the Sqns Prover.
SqnsWFOrdering → <Order Symbol> is the well founded relation that the user wants to use with the Well−Founded induction.The properties of the ordering have to be
in the knowledge base and the ordering has to be well founded.
Default value: None.

`?SqnsProofTechniques`

Option of the Sqns Prover.
SqnsProofTechniques→ <list of proof techniques >.
Proof Techniques:

ABF - arbitrary but fixed for sequences
CDLeft - Case Distinction, from the left
IndLeft - Structural Sequence Induction, from the left
WFInd - Well Founded induction (user also has to provide the well−founded relation, through the SqnsWFOrdering option)
CDRight - Case Distinction, from the right
IndRight - Structural Sequence Induction, from the right

Default value: ABF, CDLeft, IndLeft

This option specifies the order in which the proof rules should be applied on the proof situation (on the goal). As a result of this, the proof tree will be split in the corresponding branches (OR node), and each will be pursued, until one of them will succeed or all fail.

The sequence special prover is combined with several other special provers in the frame of the sequence user provers.

3.4 The Sequence User Provers

There are various combinations of provers that include also the sequence basic prover. In general a sequence user prover can be seen as improving existing provers in *Theorema* by adding sequence reasoning to them. Two groups of sequence user provers can be distinguished:

- Sequence user provers that extend predicate logic (natural deduction) provers:
  `?SqnsEqProver`
  The prover combining rewriting and proof techniques on sequences.
  `?SqnsPredProver`
  The prover combining natural deduction and proof techniques on sequences.
  `?SqnsProver`
  The prover combining natural deduction, rewriting and proof techniques on sequences.

- Sequence user provers based on the PC prover and case distinction: The PC-style prover was developed in [3], [11]. It represents an alternative to the predicate logic prover. The PC prover works in two stages:
  - a *Prove* stage, where rules for propositional/predicate logic are applied to the goal and to the non−rewrite formulae of the knowledge base, for as long as it is possible;
  - a *Compute* stage, where we apply one *generalized rewriting* (rewriting with equality, implication and equivalence as in [3], [11]) step to one formula in the proof situations.
  Another ingredient for these provers is the *case distinction* special prover, that implements proof rules for definitions by case in the language of *Theorema*.
  `?SqnsCasePC`
  The prover that integrates proof techniques for sequences and case distinction in a PC-style prover (predicate logic and generalized rewriting).
  `?SqnsEqCasePC`
  The prover that integrates proof techniques for sequences, simplification and case distinction in a PC-style prover (predicate logic and generalized rewriting).

The sequence user provers are available to the user, who will use them in the proof call:

```
Prove[ <expression>, using→<KnowledgeBase>, by→<Method>, <...>]
```

<expression> - is a *Theorema* formal text (`Proposition, Theorem, Lemma`)
<KnowledgeBase> - represents the knowledge under which the expression is to be proved. It can be provided as a list of definitions, properties, or in the frame of the `Theory` construct, provided in the language.
<Method> - is the name of a prover that is called.
<...> - there are several other parameters that can be provided, such as prover options, proof display options, etc.

4 Theory Exploration: Correctness of Merge–Sort

The formulation in the *Theorema* language of the merge–sort algorithm and its correctness was introduced in **Section 2.** The underlying theory is the theory of tuples. The methods available, the sequence provers, were described in **Section 3**. The proof of the corectness result is obtained by doing a top–down exploration, which is called "lazy thinking paradigm": deliberately don't provide any knowledge in terms of intermediate lemmata, until failing proofs of the correctness theorem (and its subtheorems) give a natural suggestion which intermediate lemmata could be useful for constructing a successful proof.

There are two ways to explore a given mathematical theory, see [4]. In the *bottom–up* approach, at each stage a new concept is introduced, then the interactions with the already defined notions are explored. These are then added to the knowledge base and the next new concept is intoduced. This exploration method yields a "complete" exploration of a given theory, the degree of completeness depending on the one doing the exploration, as this process is not automated. However, it is not very useful in this case (i.e. trying to prove a specific goal), because it involves at every stage a "complete" exploration, which means that a big part of the knowledge is not useful for proving the desired result.

In the *top–down* approach, try to prove the desired result. Chances are that the proof will fail, because not enough knowledge is available. Analyse the failed proof attempt, derive and prove conjectures needed to complete the proof. This approach is used to prove the correctness of the merge–sort algorithm.

It should be noted that in practice a combination of the two approaches is more feasible. For the purpose of this presentation, some knowledge about tuples is assumed (for example from a bottom–up exploration of the theory, which is not presented here).

4.1 Top–Down exploration: Correctness of Merge–Sort

The aim is to prove:

```
Proposition["correctness of merge-sort", any[is-tuple[A]],
     is-tuple[stmg[A]]
     stmg[A]≈ A
     is − sorted[stmg[A]]]
```

First attempt: try to prove under minimal knowledge (i.e. the notions occuring in the formulation of the theorem). The knowledge base is:

```
Theory["correctness of merge-sort:KB-1",
  Algorithm["sort by merging"]
  Algorithm["merge"]
  Definition["is-permuted-version"]
  Definition["is sorted"]]
```

The proof call:

```
Prove[Proposition["correctness of merge−sort"],
    using→Theory["correctness of merge−sort:KB-1"],
    by→SqnsEqCasePC]
```

The proof will fail. Here is the failed proof attempt (fragments of a proof generated by *Theorema*):

Prove:

(Proposition (correctness of merge−sort)) $\underset{\text{is-tuple}[A]}{\forall}(\text{is-tuple}[\text{stmg}[A]] \wedge \text{stmg}[A] \wedge \text{is-sorted}[\text{stmg}[A]])$

under the assumptions: [*list of assumptions*]

We try to prove (Proposition (correctness of merge−sort)) by making A arbitrary but fixed.

We show

(1) $\text{is} - \text{tuple}[\text{stmg}[\langle\overline{A_0}\rangle]] \wedge \text{stmg}[\langle\overline{A_0}\rangle] \approx \langle\overline{A_0}\rangle \wedge \text{is} - \text{sorted}[\text{stmg}[\langle\overline{A_0}\rangle]]$.

As there are several methods which can be applied, we have different choices to proceed with the proof.

Alternative proof 1: failed

Not all the conjunctive parts of (1)can be proved.

Proof of (1.1) $\text{is} - \text{tuple}[\text{stmg}[\langle\overline{A_0}\rangle]]$: [*details of the proof attempt*]

Proof of (1.2) $\text{stmg}[\langle\overline{A_0}\rangle] \approx \langle\overline{A_0}\rangle$: [*details of the proof attempt*]

Proof of (1.3) $\text{is} - \text{sorted}[\text{stmg}[\langle\overline{A_0}\rangle]]$ [*details of the proof attempt*]

□

The proof rule applied was "arbitrary but fixed". From analysing the failing proof attempt, we see that the proof could be completed if the following results were known:

```
Proposition["closure of merge−sort", any[is−tuple[A]],
      is−tuple[stmg[A]]]

Proposition["sorted by merge is permuted version", any[is−tuple[A]],
      stmg[A]≈A]

Proposition["sorted by merge is sorted", any[is−tuple[A]],
      is−sorted[stmg[A]]]
```

At this stage, try to prove each of the propositions. Because of space considerations, only the exploration cycle that leads to the proof of the first proposition will be presented.

4.2 Top Down Exploration Cycle: Closure of Merge–Sort

Try to prove the closure of merge–sort under a minimal knowledge base:

```
Theory["closure of merge-sort:KB 1",
  Algorithm["sort by merging"]
  Algorithm["merge"]
  Proposition["is tuple tuple"]]
```

The prove call:

```
Prove[Proposition["closure of merge−sort"],
 using→Theory["closure of merge−sort:KB 1"],built-in→Built-in["Numbers"],
   by→SqnsEqCasePC, ProofOptions→SqnsProofTechniques→"IndLeft"]
```

The proof is not successful. Analyse the failing point in the proof:

Prove:

(Proposition (closure of merge−sort)) $\underset{\text{is}-\text{tuple}[A]}{\forall} \text{is} - \text{tuple}[\text{stmg}[A]]$,

under the assumptions:

We try to prove (Proposition (closure of merge−sort)) by applying several proof methods for sequences.

Alternative proof 1: failed

We try to prove (Proposition (closure of merge−sort)) by making A arbitrary but fixed.

We show

(1) $\text{is} - \text{tuple}[\text{stmg}[\langle\overline{A_0}\rangle]]$.

As there are several methods which can be applied, we have different choices to proceed with the proof.

Alternative proof 3: failed

We try to prove (1) by case distinction using (Algorithm (sort by merging)). However, the proof fails in at least one of the cases.

Case 1:

(2) $|\langle\overline{A_0}\rangle| \leq 1$.

Hence, we have to prove

(3) $\text{is} - \text{tuple}[\langle\overline{A_0}\rangle]$.

As there are several methods which can be applied, we have different choices to proceed with the proof.

Alternative proof 1: proved

Formula (3) is proved because it is an instance of (Proposition (is tuple tuple)).

Case 2:

(4) $|\langle\overline{A_0}\rangle| \not\leq 1$.

Hence, we have to prove

(6) $\text{is} - \text{tuple}[\text{merged}[\text{stmg}[\text{lsp}[\langle\overline{A_0}\rangle]], \text{stmg}[\text{rsp}[\langle\overline{A_0}\rangle]]]]$.

As there are several methods which can be applied, we have different choices to proceed with the proof.

[*proof fails*]

□

The point where the proof is stuck can be surpassed if the following holds:

```
Proposition["closure of merge, with splits", any[is−tuple[A]],
     is−tuple[merged[lsp[A], rsp[A]]]]
```

This is a closure property, and it represents the interaction of the closure properties of the merge algorithm. Closure is a basic property in the theory of tuples. So a bottom−up exploration of the theory will contain the closure properties of splits and merge, as described above. With this additional knowledge, the proof of the proposition is proved easily (details will be skipped).

However, trying to prove the closure of merge−sort with using arbitrary but fixed, case distinction or structural induction fails, even with this additional knowledge. Now try to prove it using well−founded induction. As describes in **Section 3**, the user has to provide the well−founded ordering to the prover. The user has to make sure that the ordering provided is well−founded. The natural well−ordering for tuples is "is−longer−tuple", and the its properties are proved in a bottom−up exploration of the theory.

The knowledge base under which the proof is attempted:

```
Theory["closure of merge-sort:KB 3",
  Algorithm["sort by merging"]
  Algorithm["merge"]
  Definition["is-permuted-version"]
  Proposition["is tuple tuple"]
  Definition["length of tuples"]
  Proposition["numbers"]
  Theory["well-founded relations on tuples"]]
```

The proof call:

Prove[Proposition["closure of merge−sort"],
using→Theory["closure of merge−sort:KB 3"],
built−in→Built−in["Numbers"] , by→SqnsEqCasePC,
ProverOptions→ $\{SqnsProofTechniques \to \{"WFInd"\}, SqnsWFOrdering \to "\succ" \}$]

The proof attempt:

Prove:

(Proposition (closure of merge−sort)) $\underset{\text{is}-\text{tuple}[A]}{\forall} \text{is}-\text{tuple}[\text{stmg}[A]]$,

under the assumptions:

We try to prove (Proposition (closure of merge−sort)) by well−founded induction on A .

Well−founded induction:

Well−Founded Induction Hypothesis:

(1) $\underset{\text{is}-\text{tuple}[x1]}{\forall} (\langle\overline{A_0}\rangle \succ x1 \Rightarrow \text{is}-\text{tuple}[\text{stmg}[x1]])$

We have to show:

(2) $\text{is}-\text{tuple}[\text{stmg}[\langle\overline{A_0}\rangle]]$.

As there are several methods which can be applied, we have different choices to proceed with the proof.

Alternative proof 3: failed

We try to prove (2) by case distinction using (Algorithm (sort by merging)). However, the proof fails in at least one of the cases.

Case 1:

(3) $|\langle\overline{A_0}\rangle| \leq 1$.

Hence, we have to prove

(4) $\text{is}-\text{tuple}[\langle\overline{\text{A}_0}\rangle]$.

As there are several methods which can be applied, we have different choices to proceed with the proof.

Alternative proof 1: proved

Formula (4) is proved because it is an instance of (Proposition (is tuple tuple)).

Case 2:

(5) $|\langle\overline{\text{A}_0}\rangle| \not\leq 1$.

Hence, we have to prove

(7) $\text{is}-\text{tuple}[\text{merged}[\text{stmg}[\text{lsp}[\langle\overline{\text{A}_0}\rangle]],\text{stmg}[\text{rsp}[\langle\overline{\text{A}_0}\rangle]]]]$.

As there are several methods which can be applied, we have different choices to proceed with the proof.

[*failed branches*]

□

Formula (7) is an instance of `Proposition["closure of merge, with splits"]`, if `is−tuple[stmg[lsp[`$\langle\overline{\text{A}_0}\rangle$` ]]]`, `is−tuple[stmg[rsp[`$\langle\overline{\text{A}_0}\rangle$` ]]]`. But this can be obtained by modus ponens from the well−founded induction hypothesis, knowing that splits are shorter than the tuple. This information we take from the bottom−up exploration of the theory. The proof goes through (details will be skipped).

In a similar way, the proofs of `Proposition["sorted by merge is permuted version"]` and `Proposition["sorted by merge is sorted"]` are obtained in a few exploration cycles (details will be skipped). This completes the proof of the corectness of the merge−sort algorithm.

5 Conclusions and future work

Although the example chosen here is quite simple, it demonstrates to be a good testbed for the development of the sequence provers in *Theorema*. It illustrates the capabilities of the system, notably the language and the proofs in natural style, generated automatically. The problem is not new (it was addressed for instance in [9], [10]), but the formulation in a theory with sequences makes use of powerful sequence reasoning mechanisms, such as sequence unification ([7]) (needed in the case of well−founded induction, in order to use the induction hypothesis).

To complete the proof, a series of exploration rounds were carried out. The natural style of the proofs make it easy for the user to derive new conjectures from the failed attempts. Moreover, as shown in [4], this process can be automated, by designing a proof analyser that detects the failing points of a proof and generates new properties, which it will attempt to prove. In this way, more complicated examples can be addressed. This is one of the long−term goals of the *Theorema* project.

On a shorter term, work is being carried out to improve the interaction between the sequence prover and the rest of the basic provers available in the system, and to better integrate the proving activity with solving and computing.

References

[1] B.Buchberger; Case Study: Exploration of Sorting by Merging; *Mathematica notebooks, part of a lecture for high−school teachers, RISC-Linz, October 2001*; not published.

[2] B.Buchberger; Mathematica as a Rewrite Language ; *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming), November 1-4, 1996*; pp.1-13.

[3] B. Buchberger; The PCS Prover in Theorema; *Proceedings of EUROCAST 2001 (8th International Conference on Computer Aided Systems Theory - Formal Methods and Tools for Computer Science), Feb. 19-23, 2001, Las Palmas de Gran Canaria, (R. Moreno-Diaz, B. Buchberger, J.L. Freire eds.)*; Lecture Notes in Computer Science 2178, 2201, pp. 469-478.

[4] B. Buchberger; Theory Exploration with Theorema; *Proceedings of SYNASC 2000, Oct. 4-6, 2000, Timisoara, Romania*; (T. Jebelean, V. Negru, A. Popovici eds.); Analele Universitatii Din Timisoara, Ser. Matematica-Informatica, Vol. XXXVIII, Fasc.2, 2000; pp. 9-32.

[5] B.Buchberger and D.Vasaru; An Induction Prover for Equalities over Lists; *Proceedings of the First Theorema Workshop, Hagenberg, Austria, June 9-10, 1997*; RISC-Report Series No.97-20, 1997.

[6] B.Buchberger, C.Dupre, T.Jebelean, K.Kriftner, K.Nakagawa, D.Vasaru and W.Windsteiger; The Theorema Project: A Progress Report; *In M.Kerber and M.Kolhase, editors, Proceedings of Calculemus 2000: Integration of Symbolic Computation and Mechanized Reasoning*; A.K. Petersm Natick, Massatchussets, 2000, pp.98-113.

[7] T.Kutsia; Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols; *PhD Thesis. Research Institute for Symbolic Computation, Johannes Kepler University. Linz, Austria*, 2002.

[8] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill Book Company, 1974.

[9] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming*, Volume 1: Deductive Reasoning, Addison-Wesley Publishing Company,Inc. 1985.

[10] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming*, Volume 2: Deductive Systems, Addison-Wesley Publishing Company,Inc. 1990.

[11] D.Vasaru−Dupre; Automated Theorem Proving by Integrating Proving, Solving and Computing;*PhD Thesis The Research Institute for Symbolic Computation, Johannes Kepler University*; May 2000; RISC report 00-19.

A Appendix

This appendix contains the definitions and properties of notions relevant to the proofs in the paper. They are given in the order they appear in the bottom−up exploration of the theory:

Proposition["is tuple tuple",any[$\overline{x}$], $is-tuple[\langle\overline{x}\rangle]$]

Definition["append",any[x,$\overline{y}$], $\langle\overline{y}\rangle \frown x = \langle\overline{y}, x\rangle$]

Proposition["append empty",any[x], $\langle\rangle \frown x = \langle x\rangle$]

Proposition["closure of append", any[is−tuple[X], y], is−tuple[X$\frown$y]]

Definition["prepend",any[x,$\overline{y}$], $x \smile \langle\overline{y}\rangle = \langle x, \overline{y}\rangle$]

Proposition["prepend empty",any[x], x$\smile$ $\langle\rangle = \langle x\rangle$]

Proposition["Closure of prepend",any[a,is−tuple[X]], is−tuple[a$\smile$X]]

Definition["Membership",any[x,y,$\overline{y}$],

$$x \notin \langle\rangle$$
$$(x \in \langle y, \overline{y}\rangle) \quad \Leftrightarrow ((x = y) \vee x \in \langle\overline{y}\rangle)(0)$$

Proposition["element with prepend",any[x,y,is−tuple[X]],(x$\in$X)$\Rightarrow$(x$\in$y$\smile$X)]

Definition["Deletion of the First Occurrence",any[a,x,$\overline{x}$],

$$dfo[a,\ \langle\rangle] = \langle\rangle$$
$$dfo[a,\ ,\overline{x}\rangle\ ] \ = \begin{array}{lll} \langle\overline{x}\rangle & \Leftarrow & x = a \\ x[a,\ \langle\overline{x}\rangle] & \Leftarrow & otherwise \end{array} \quad]$$

Proposition["Closure of deletion",any[a,is−tuple[X]], is−tuple[dfo[a,X]]]

Definition["is−permuted−version",any[x,$\overline{x}, y, \overline{y}$],

$$\langle\rangle \approx \langle\rangle$$
$$\langle\rangle \not\approx \langle y, \overline{y}\rangle$$
$$\langle x, \overline{x}\rangle \approx \langle\overline{y}\rangle \quad \Leftrightarrow (x \in \langle\overline{y}\rangle \wedge \langle\overline{x}\rangle \approx dfo[x,\ \langle\overline{y}\rangle])]$$

Definition["length of tuples",any[x,$\overline{x}, \overline{y}$],

$$|\langle\ \ \rangle| = 0$$
$$|\langle\ x, \overline{y}\rangle| = |\langle\ \ \overline{y}\rangle|^{+}\ \ ]$$

Definition["Is-Longer-Than",any[x,$\overline{x}, y, \overline{y}$],

$$\neg\langle\rangle \succ \langle\overline{y}\rangle$$
$$\langle x, \overline{x}\rangle \succ \langle\rangle$$
$$\langle x, \overline{x}\rangle \succ \langle y, \overline{y}\rangle \quad \Leftrightarrow \langle\overline{x}\rangle \succ \langle\overline{y}\rangle \quad]$$

Definition["is sorted",any[$\overline{x}, x, y, \overline{y}$],

$$is-sorted[\langle\rangle]$$
$$is-sorted[\langle x\rangle]$$
$$is-sorted[\langle\overline{x}, x, y, \overline{y}\rangle] \quad \Leftrightarrow (\wedge \begin{array}{l} is-sorted[\langle\overline{x}, x\rangle] \\ x \geq y \\ is-sorted[\langle y, \overline{y}\rangle] \end{array} \quad)\ ]$$

Definition["left split", any[a, b, $\overline{x}$],

$$lsp[\langle\rangle] := \langle\rangle$$
$$lsp[\langle a\rangle] := \langle a\rangle$$

$lsp[\langle a,\ \overline{x},\ b\rangle := a \smile lsp[\overline{x}]]$]

Definition["right split", any[a, b, $\overline{x}$],

$rsp[\langle\rangle] := \langle\rangle$

$rsp[\langle a\rangle] := \langle\rangle$

$rsp[\langle a,\ \overline{x},\ b\rangle := rsp[\overline{x}]$]]

Proposition["closure of splits", any[is-tuple[A]],

is-tuple[lsp[A]]

is-tuple[rsp[A]]]

Proposition["splits are shorter", any[is-tuple[A]],

A $\succ lsp[A]$

$A \succ rsp[A]$]

Algorithm["merge", any[is-tuple[A, B], a, b, $\overline{x}$, $\overline{y}$],

$merged[\langle\rangle,\ A] := A$

$merged[B,\ \langle\rangle] := B$

$merged[\langle a,\ \overline{x}\rangle,\ ,\ \overline{y}\rangle] := \begin{array}{lll} a \smile merged[\langle\overline{x}\rangle,\ \langle b,\ \overline{y}\rangle] & \Leftarrow & a \geq b \\ b \smile merged[\langle a, \overline{x}\rangle,\ \langle\overline{y}\rangle] & \Leftarrow & otherwise \end{array}$]

Proposition["closure of merge", any[is−tuple[A, B]], is−tuple[merged[A, B]]]

Proposition["merge splits yields permuted version", any[is−tuple[A]],

A≈merged[lsp[A], rsp[A]]]