Institute e-Austria in Timisoara

IeAT Report Series

Algorithm Synthesis by Lazy Thinking - Examples and Implementation in *Theorema*

Bruno BUCHBERGER, Adrian CRACIUN

2003

Algorithm Synthesis by Lazy Thinking: Examples and Implementation in *Theorema*

Bruno Buchberger [1] and Adrian Crăciun [2]

RISC - Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria

Abstract

Recently, we proposed a systematic method for top-down synthesis and verification of lemmata and algorithms called "lazy thinking method" as a part of systematic mathematical theory exploration (mathematical knowledge management). The lazy thinking method is characterized:

- by using a library of *theorem and algorithm schemes*
- and by using the information contained in *failing attempts to prove the schematic theorem or the correctness theorem for the algorithm scheme* for *inventing lemmata or requirements for subalgorithms*, respectively.

In this paper, we give a couple of examples for algorithm synthesis using the lazy thinking paradigm. These examples illustrate how the synthesized algorithm depends on the algorithm scheme used. Also, we give details about the implementation of the lazy thinking algorithm synthesis method in the frame of the *Theorema* system. In this implementation, the synthesis of the example algorithms can be carried out completely automatically, i.e. without any user interaction.

Key words: algorithm invention, algorithm verification, program synthesis, algorithm correctness, re-usable algorithms, algorithm schemes, learning from failure, conjecture generation, lazy thinking, requirement engineering, didactics of programming, mathematical knowledge retrieval, mathematical knowledge management, sorting, merging, merge-sort, *Theorema.*

[1] Sponsored by FWF (Österreichischer Fonds zur Förderung der Wissenschaftlichen Forschung; Austrian Science Foundation), Project SFB 1302, in the frame of the SFB "Scientific Computing" at the Johannes Kepler University, Linz, Austria. Email: buchberger@risc.uni-linz.ac.at

[2] Sponsored by the Calculemus European Project. Email: acraciun@risc.uni-linz.ac.at

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

The lazy thinking approach to the synthesis of correct algorithms was introduced by the first author in various talks since 2001, see the proceedings paper [5]. The lazy thinking approach to algorithm synthesis is an adaptation of the lazy thinking paradigm for lemmata invention introduced in [4]. Lemmata invention is part of a general philosophy of systematic theory exploration, see [4], as an alternative to the isolated theorem proving paradigm that prevailed for many decades in the area of automated theorem proving. Systematic theory exploration is one of the main topics in the emerging field of "mathematical knowledge management" initiated in September 2001 by the 1st International Workshop on Mathematical Knowledge Management (MKM), see [7] and [10], which engendered MKM 2003 [2] and also parallel events in North America, see [11].

Roughly, our lazy thinking algorithm synthesis method proceeds as follows:

- We start with a (predicate logic) specification of the algorithm in a particular data domain. We assume that all the auxiliary operations (functions and predicates) occurring in the specification are defined and all important properties of these operations are already known, i.e. that the "theory of these operations" is completely explored.
- The method now tries out, one after the other, various "algorithm schemes" that are stored in a library of algorithm schemes for the given mathematical domain. An algorithm scheme is a predicate logic formula that describes an algorithm (recursively) in terms of unspecified subalgorithms together with a proof method appropriate for (induction) proofs of properties of algorithms following this scheme.
- For the chosen algorithm scheme, the proof method is called for proving the correctness theorem, i.e. for proving that the unknown algorithm satisfies the given specification. Typically, this proof will fail because nothing is known about the unspecified subalgorithms.
- From the failing proof situation, by a conjecture generating algorithm, "requirements" (specifications) for the unknown subalgorithms are generated that enable the prover to complete the proof successfully. These requirements are added to the knowledge base and the proof of the correctness theorem is attempted again. Now, the proof will get over the failing situation and will either succeed or will fail again at some later proof situation.
- This procedure is iterated in a recursive cascade until the proof of the correctness theorem goes through (or one gives up). After successful termination, the following will be true: Under the assumption that all ingredient subalgorithms satisfy the requirements generated, the main algorithm satisfies the problem specification.
- In this stage, there are two possibilities: Either, in the initial knowledge base, algorithms are available that satisfy the requirements for the subalgo-

rithms described in the lemmata and we are done, i.e. a correct algorithm has been synthesized for the initial problem and its correctness proof has been generated. Or subalgorithms that satisfy the requirements can be synthesized by another application of the same method in a next round of the procedure.

The distinctive features of our algorithm synthesis method, as compared to other methods, are:

- the use of algorithm schemes taken from a library of algorithm schemes,
- the crucial role of failing proofs and conjecture generation from failing proofs,
- the decomposition of theory exploration and, in particular, algorithm invention and verification into theory layers,
- the naturalness of the approach, which makes it attractive both for complete or partial automation in computer-supported systems for formal mathematics and also for usage as a strategy for human algorithm invention and teaching. (In fact, the idea for the lazy thinking paradigm for algorithm synthesis arose while the first author was preparing a course on mathematical algorithm verification for high-school teachers in October 2001.)

Similar ideas were introduced in synthesis literature (see [3] for an overview). However, in one respect or the other, our synthesis method is different from each of the above. In the following, we go through a few of these.

The deductive synthesis method, of the type described in [1],[14] use proof planning to set up the proof of the correctness of an algorithm, replaces the unknown parts of the algorithm (existential parts of the specification) with metavariables which will be instantiated as the proof planning progresses.

In formal methods, starting from a specification, by a series of transformations a correct program is reached. Adding to this setting, additional knowledge about algorithm development (data abstractions, design abstractions - that correspond to algorithm schemes such as divide-and-conquer [16]) has led to the implementation of powerful systems, such as KIDS [17] and its successor Specware [19] that support the use of schemes for the synthesis of programs.

In the same spirit, schemas of logic programs, with added semantic meaning, are transformed into programs by a series of steps that preserve correctness in [13].

In (automated) software engineering, there are already catalogues of software design patterns available, which again, accumulate knowledge on software development (”recurring solutions to standard problems”, see [15]). These are directed towards specific object-oriented or concurrent, parallel and distributed programming languages (for pattern resources, see [20]).

The lazy thinking algorithm synthesis method is independent of any system for formal mathematics. However, the chosen system must have a couple of properties in order to qualify as a frame for the method:

- The proof objects generated by the automated theorem proving part of the system must be open for post-processing. In other words, black-box provers are not suitable for the method because the important (automated) requirement generation for subalgorithms is based on a detailed analysis of the proof objects generated.
- For the same reason, it is also crucial that the automated theorem provers used in the system yield proof objects also in the case that proofs fail. In distinction to other algorithm synthesis methods, which extract algorithms from successful proofs, the essence of our method is the automated generation of requirements for subalgorithms from failing proofs!
- The automated theorem provers used in the system should be ”natural style” provers, e.g. natural deduction provers or variants thereof: The generation of requirements for the unknown subalgorithms is crucially based on the analysis of the temporary assumptions and temporary proof goals in the failing proof object.

In this paper, all our formal developments will be given in the *Theorema* system, see [8], [9]. In particular, all formulae will be given in the *Theorema* syntax. Also, the implementation of the method is described in the concrete example of the *Theorema* system.

This paper consists of three case studies of algorithm synthesis and remarks on the implementation of the method in the frame of *Theorema*, which was carried out by the second author based on an earlier version of the *Theorema* induction prover and conjecture generator by the first author. The first case study, sorting by merging, is the one of [5], which we summarize here because it is best suited for explaining the general method. The other two case studies are new. The second case study (synthesis of a merge algorithm) illustrates particularly well that, although finding the induction proofs and the requirements for the subalgorithms is quite easy, keeping track of the organization and the details of these proofs and requirements is a quite demanding task that needs automation for making the entire method practically attractive. The third case study shows how, for a fixed algorithm specification, changing the algorithm scheme fed into the synthesis procedure changes the algorithm synthesized, i.e. the specification (requirements) generated automatically for the subalgorithms.

2 First Case Study: Sorting by Merging

2.1 Problem Specification

We want to synthesize an algorithm ’*sorted*’ that meets the following specification:

$$\underset{\mathit{is\text{–}tuple}[X]}{\forall} \mathit{is\text{–}sorted\text{–}version}[X, \mathit{sorted}[X]],$$

where

$$\underset{is\text{–}tuple[X]}{\forall}\left(is\text{–}sorted\text{–}version[X,Y] \Leftrightarrow \begin{cases} is\text{–}tuple[Y] \\ X \approx Y \\ is\text{–}sorted[Y] \end{cases}\right).$$

The predicate '*is-sorted-version*' is defined in terms of the two auxiliary predicates '≈' and '*is-sorted*'. (For $X \approx Y$ read 'Y is a permuted version of X' or 'X and Y contain the same elements equally often'):

$$is\text{–}sorted[\langle\rangle],$$

$$\underset{x}{\forall} is\text{–}sorted[\langle x\rangle],$$

$$\underset{x,y,\bar{z}}{\forall}\left(is\text{–}sorted[\langle x,y,\bar{z}\rangle] \Leftrightarrow \begin{cases} x \geq y \\ is\text{–}sorted[\langle y,\bar{z}\rangle] \end{cases}\right)$$

and

$$\langle\rangle \approx \langle\rangle,$$

$$\underset{y,\bar{y}}{\forall}\langle\rangle \not\approx \langle y,\bar{y}\rangle,$$

$$\underset{x,\bar{x},\bar{y}}{\forall}(\langle x,\bar{x}\rangle \approx \langle\bar{y}\rangle) \Leftrightarrow (x \in \langle\bar{y}\rangle \wedge \langle\bar{x}\rangle \approx dfo[x,\langle\bar{y}\rangle])) \ .$$

(For the "sequence variables" notation $\bar{x}$ etc., see the papers on *Theorema*, e.g. [8]. Sequence variables can be replaced by arbitrarily many terms. Thus, for example, the terms $\langle\rangle$, $\langle 2\rangle$, $\langle 4,3,2,2\rangle$, $\langle 4,y,a,z,a+b\rangle$ are instances of the term $\langle\bar{x}\rangle$. We use angle brackets as constructors for tuples: for example, $\langle 2,2,3,1,4\rangle$ is the tuple consisting of the elements 2, 2, 3, 1, 4.)

The definitions of '*is-sorted*' and '≈', again, contain auxiliary operations like '∈' (read: 'is element') and 'dfo' (read: 'delete first occurrence') that must be defined in terms of other auxiliary functions until we arrive at the basic operations on tuples. The definitions of all these auxiliary operations and also the formulae describing various properties of these auxiliary operations are supposed to be contained in the knowledge base, see appendix.

(In [5] we discuss the question of "completeness" and "sufficiency" of knowledge bases.)

2.2 Algorithm Scheme

In our view, an "algorithm scheme" (or "algorithm type") for a given data domain,

- is a recursive definition of an unspecified "main" operation in terms of other unspecified "auxiliary" operations and the basic operations of the data domain,
- together with a proof method that corresponds to the recursive definition and the data domain in a natural way.

In our case (a problem on the data type of tuples), a possible recursive definition of the solution function is the well-known "divide–and–conquer" scheme (in a version which is appropriate to the data type of tuples):

$$\underset{\mathit{is\text{–}tuple}[X]}{\forall} \mathit{sorted}[X] = \begin{cases} \mathit{special}[X] & \Leftarrow \mathit{is\text{–}trivial\text{–}tuple}[X] \\ \mathit{merged}[\mathit{sorted}[\mathit{left\text{–}split}[X]], & \\ \qquad \mathit{sorted}[\mathit{right\text{–}split}[X]]] & \text{otherwise} \end{cases}$$

where you should think about the "main function" '*sorted*' and the "auxiliary functions" '*special*', '*merged*', '*left–split*', and '*right–split*' as completely unspecified (except that '*sorted*' is related to the auxiliary functions as described in the scheme). In fact, at this moment, nothing is known about these functions that would justify to give them names like '*sorted*', '*special*', '*merged*' etc.

Note also, that in contrast to the operations '*sorted*' etc., the predicate '*is-trivial-tuple*' is not unspecified but, rather, is defined by a formula in the knowledge base, see the appendix.

Also, we include the following natural "type requirements" on the auxiliary functions as a part of the algorithm scheme:

$$\underset{\substack{\mathit{is\text{–}tuple}[X] \\ \mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \mathit{is\text{–}tuple}[\mathit{special}[X]],$$

$$\underset{\substack{\mathit{is\text{–}tuple}[X] \\ \neg\mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \begin{cases} \mathit{is\text{–}tuple}[\mathit{left\text{–}split}[X]] \\ \mathit{is\text{–}tuple}[\mathit{right\text{–}split}[X]] \end{cases},$$

$$\underset{\mathit{is\text{–}tuple}[Y,Z]}{\forall} \mathit{is\text{–}tuple}[\mathit{merged}[Y,Z]] \; .$$

The type requirements will be important for being able to prove that the function '*sorted*', for tuple arguments, yields tuples as results.

Finally, we also consider the following requirement

$$\underset{\substack{\mathit{is\text{–}tuple}[X] \\ \neg\mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \begin{cases} X \succ \mathit{left\text{–}split}[X] \\ X \succ \mathit{right\text{–}split}[X] \end{cases}$$

as a part of the recursive definition, which guarantees termination of the algorithm in case '$\succ$' is a Noetherian predicate. (In our case, we will use the predicate 'has shorter length' for '$\succ$', see the definition in the appendix.)

Second, we include the following special induction method into the algorithm scheme:

- In order to prove, for an arbitrary property A, $\underset{\mathit{is\text{–}tuple}[X]}{\forall} A[X]$ it suffices to prove, for an arbitrary but fixed $\bar{x_0}$, $A[\langle\bar{x_0}\rangle]$ under the assumptions $\mathit{is\text{–}tuple}[\langle\bar{x_0}\rangle]$ and $\underset{\substack{\mathit{is\text{–}tuple}[Y] \\ \langle\bar{x_0}\rangle \succ Y}}{\forall} A[Y]$.

This particular induction method is based on the property that $\succ$ is a Noetherian relation.

One might argue that, with the inclusion of an appropriate inductive proof method into the algorithm scheme, already very much of the "invention" is taken away from the automated invention system. However, in future mathematical knowledge management systems (and, in particular, verified algorithm invention systems), it would be silly to throw away the accumulated knowledge of mathematicians on problem solving "schemes". Rather, in future systems, the accumulated algorithm invention knowledge of mathematicians should be kept available in "algorithm schemes libraries" that can then be used, in the way which we demonstrate in this paper, for inventing concrete algorithms for concrete problems.

A completely automatic search through a library of possible algorithm schemes to be applied for a particular algorithm synthesis problem may seem to be combinatorially prohibitive. However, in practice, we think that our strategy is viable for the following reasons:

- Given a data type (which is part of the problem specification), experience shows that there are not too many possible algorithm schemes that are worth storing in a library of algorithm scheme. There are many algorithms but, in contrast, there are only a few algorithm schemes. In other words, given a data type, there are only of few basic ideas how to attack problems specified for the given data type but there are many variations or instantiations of these ideas and even more combinations of theses ideas with ideas (schemes) for the subalgorithms.
- Of course, not all algorithm schemes in the library of algorithm schemes for a given data type will lead to reasonable algorithms for the problem specified. More concretely, it will turn out the requirements for the subalgorithms derived from analyzing failing proofs of the respective correctness theorem may not be easily satisfiable or not be satisfiable at all in the next round of the synthesis procedure. We guess that some user interaction will be necessary to follow only promising paths in the overall synthesis procedure through a couple of layers of subalgorithms, subsubalgorithms etc. However, at the moment, we do not yet have sufficiently much experimental material from case studies in order to make a well-founded statement on the amount of user interaction necessary or sufficient in our approach.

2.3 Algorithm Synthesis by Lazy Thinking: First Round

We now start from the following situation:

- We have a knowledge base consisting of all the definitions and essential properties of the operations and auxiliary operations (functions and predicates) occurring in the problem specification (in our case: the specification of the binary predicate '*is–sorted–version*' and all auxiliary operations, like '$\succ$' etc., see appendix).

- We have chosen an algorithm scheme (including an induction scheme) from a finite library of algorithm schemes for the domain of tuples (in our case: the "divide–and–conquer" algorithm scheme; in the third case study we will start from some other scheme!).

We do now the following:

- We include the algorithm scheme for '*sorted*', the type requirements for the auxiliary functions, and the requirements on the decreasing length of '*left-split*' and '*right-split*' into the knowledge base.
- Then we start attempting to prove the correctness theorem

$$\underset{\textit{is–tuple}[X]}{\forall} \textit{is–sorted–version}[X, \textit{sorted}[X]].$$

- Of course, this proof will not succeed because, at this moment, essentially nothing is known about the auxiliary functions '*merged*', '*left–split*' etc. We proceed with the proof until the proof gets stuck.
- When it got stuck, we analyze the current, failing proof situation and try to conjecture requirements (properties) of the auxiliary subalgorithms '*special*', '*left-split*', '*right–split*', '*merged*' that would make it possible to get over the failing proof situation. (It is essential for the automation of the method that this step of conjecturing suitable requirements can be automated!)
- We add the conjectured requirements to the knowledge base and repeat the whole process, i.e. we go to the next round in the algorithm invention process.

In the example, the failing proof attempt (which can be generated completely automatically by the *Theorema* induction prover) is as follows:

Proof Attempt Begin

For proving the correctness theorem, we use well-founded induction w.r.t. $\succ$ on X:

We assume *is–tuple*$[\langle \bar{x_0} \rangle]$ and the induction hypothesis

$$\underset{\substack{\textit{is–tuple}[Y] \\ \langle \bar{x_0} \rangle \succ Y}}{\forall} \textit{is–sorted–version}[Y, \textit{sorted}[Y]]$$

and we show

$$\textit{is–sorted–version}[\langle \bar{x_0} \rangle, \textit{sorted}[\langle \bar{x_0} \rangle]].$$

We use the algorithm scheme for '*sorted*' and distinguish two cases:

CASE *is-trivial-tuple*$[\langle \bar{x_0} \rangle]$:

In this case, we have to show

$$\textit{is–sorted–version}[\langle \bar{x_0} \rangle, \textit{special}[\langle \bar{x_0} \rangle]]$$

i.e., by the definition of '*is-sorted-version*', we have to show

$$is\text{–}tuple[special[\langle \bar{x_0}\rangle]] \quad (G1)\ ,$$

$$special[\langle \bar{x_0}\rangle] \approx \bar{x_0} \qquad (G2)\ ,$$

$$is\text{–}sorted[special[\langle \bar{x_0}\rangle]] \ (G3)\ .$$

For $(G2)$, by the fact that

$$\underset{is\text{–}trivial\text{–}tuple[X],is\text{–}tuple[Y]}{\forall}((X \approx Y) \Leftrightarrow (X = Y))$$

it suffices to prove that $special[\langle \bar{x_0}\rangle] = \langle \bar{x_0}\rangle$.

$(G1)$ is true by the type requirement on special. We can not prove $(G2)$ and $(G3)$.
Proof Attempt End

(The proof attempt generated automatically by the *Theorema* induction prover for tuples is basically exactly like the proof attempt above including the explanatory English text, see the papers on *Theorema.* However, for the presentation in this paper, we leave out some intermediate steps of the *Theorema* proofs.)

Now we analyze the failing proof situation and find:

- We have the case assumption as the only temporary assumption:

$$is\text{–}trivial\text{–}tuple[\langle \bar{x_0}\rangle].$$

- We have the temporary goal:

$$special[\langle \bar{x_0}\rangle] = \langle \bar{x_0}\rangle.$$

It is obvious to conjecture (and our current *Theorema* conjecture generating algorithm can do this automatically) that the following requirement on the function '*special*'

$$\underset{is\text{–}trivial\text{–}tuple[X]}{\forall}(special[X] = X)$$

will make it possible to get over the failing proof situation.

We add this requirement to the knowledge base and proceed to the next invention round.

2.4 Algorithm Synthesis by Lazy Thinking: Second Round

Since we have added a requirement on the auxiliary function '*special*' we will get now over the failing proof situation and we will be stuck at some later situation in the proof in which, again, we will try to invent a requirement on the auxiliary functions that will make it possible to proceed further.

The next proof attempt (which again can be generated completely automatically by the *Theorema* induction prover) will now proceed as follows:

PROOF ATTEMPT BEGIN

CASE $\neg$*is-trivial-tuple*$[\langle \bar{x_0} \rangle]$:

In this case, we have to show

$$\textit{is–sorted–version}[\langle \bar{x_0} \rangle, \textit{merged}[\textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]], \textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]]]].$$

For this, by the definition of '*is-sorted-version*', it suffices to show

$$\textit{is–tuple}[\textit{merged}[\textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]], \textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]]]] \quad (H1),$$

$$\langle \bar{x_0} \rangle \approx \textit{merged}[\textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]], \textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]]] \quad (H2),$$

$$\textit{is–sorted}[\textit{merged}[\textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]], \textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]]]] \quad (H3).$$

From the case assumption, by the type requirements on '*left-split*' and '*right-split*', the property that '*left-split*' and '*right-split*' produce shorter tuples, and the induction hypothesis we obtain

$$\textit{is–sorted–version}[\textit{left–split}[\langle \bar{x_0} \rangle], \textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]]],$$

$$\textit{is–sorted–version}[\textit{right–split}[\langle \bar{x_0} \rangle], \textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]]].$$

From this, by the definition of '*is-sorted-version*', we obtain

$$\textit{is–tuple}[\textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]]] \quad (AL1),$$

$$\textit{left–split}[\langle \bar{x_0} \rangle] \approx \textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]] \quad (AL2),$$

$$\textit{is–sorted}[\textit{sorted}[\textit{left–split}[\langle \bar{x_0} \rangle]]] \quad (AL3),$$

$$\textit{is–tuple}[\textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]]] \quad (AR1),$$

$$\textit{right–split}[\langle \bar{x_0} \rangle] \approx \textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]] \quad (AR2),$$

$$\textit{is–sorted}[\textit{sorted}[\textit{right–split}[\langle \bar{x_0} \rangle]]] \quad (AR3).$$

$(H1)$ follows from $(AL1)$ and $(AR1)$ by the type requirement on '*merged*'.

We cannot prove $(H2)$ and $(H3)$.

PROOF ATTEMPT END

Now we analyze the failing proof situation and find:

- We have the case assumption and the formulae $(AL1)$, ..., $(AR3)$ as temporary assumptions.
- We have the temporary goals $(H2)$ and $(H3)$.

It is not so near at hand but, after some thinking, relatively easy to conjecture (and our current *Theorema* conjecture generating algorithm can produce this conjecture automatically) that the following requirement on the functions '*left–split*', '*right–split*' and '*merged*'

$$\underset{\substack{is\text{–}tuple[X,Y,Z]\\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall}\left(\begin{cases} left\text{–}split[X] \approx Y \\ right\text{–}split[X] \approx Z \\ is\text{–}sorted[Y] \\ is\text{–}sorted[Z] \end{cases} \Rightarrow \begin{cases} merged[Y,Z] \approx X \\ is\text{–}sorted[merged[Y,Z]] \end{cases}\right)$$

will make it possible to get over the failing proof situation.

We add this requirement to the knowledge base and now (H2) and (H2) can be proved and the entire proof can be completed.

2.5 Result of Synthesis

If we now collect the requirements on the functions '*special*','*left–split*','*right–split*' and '*merged*', we see that we invented and proved the following "Relative Correctness Theorem for Merge-Sort":

$$Knowledge[is\text{-}sorted\text{-}version] \quad \Rightarrow$$
$$\underset{special,merged,left\text{–}split,right\text{–}split}{\forall}\left(\begin{array}{c} Is\text{-}Merge\text{-}Sort\text{-}Algorithm[sorted, special, merged, \\ \qquad left\text{-}split, right\text{-}split] \\ \Rightarrow \\ \underset{is\text{–}tuple[X]}{\forall} is\text{–}sorted\text{–}version[X, sorted[X]] \end{array}\right).$$

Here, '*Knowledge*' is the conjunction of all (some of the) formulae known about the predicate '*is-sorted-version*' and all its ingredient operations (functions and predicates), like '*is-sorted*', '≈', etc. (see appendix) and '*Is–Merge–Sort–Algorithm*' is defined as follows:

$$\underset{special,merged,left\text{–}split,right\text{–}split}{\forall}\left(\begin{array}{c} Is\text{-}Merge\text{-}Sort\text{-}Algorithm[sorted, special, \\ \qquad merged, left\text{-}split, right\text{-}split] \end{array}\right)$$
$$\Leftrightarrow$$

$$\left\{\begin{array}{l}
\underset{\mathit{is\text{–}tuple}[X]}{\forall} \mathit{sorted}[X] = \begin{cases} \mathit{special}[X] & \Leftarrow \mathit{is\text{–}trivial\text{–}tuple}[X] \\ \mathit{merged}[\mathit{sorted}[\mathit{left\text{–}split}[X]], & \\ \qquad \mathit{sorted}[\mathit{right\text{–}split}[X]]] & \text{otherwise} \end{cases} \\
\underset{\mathit{is\text{–}trivial\text{–}tuple}[X]}{\forall} (\mathit{special}[X] = X) \\
\underset{\substack{\mathit{is\text{–}tuple}[X] \\ \neg \mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \begin{cases} \mathit{is\text{–}tuple}[\mathit{left\text{–}split}[X]] \\ X \succ \mathit{left\text{–}split}[X] \\ \mathit{is\text{–}tuple}[\mathit{right\text{–}split}[X]] \\ X \succ \mathit{right\text{–}split}[X] \end{cases} \\
\underset{\mathit{is\text{–}tuple}[Y,Z]}{\forall} \mathit{is\text{–}tuple}[\mathit{merged}[Y,Z]] \\
\underset{\substack{\mathit{is\text{–}tuple}[X,Y,Z] \\ \neg \mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \left(\begin{cases} \mathit{left\text{–}split}[X] \approx Y \\ \mathit{right\text{–}split}[X] \approx Z \\ \mathit{is\text{–}sorted}[Y] \\ \mathit{is\text{–}sorted}[Z] \end{cases} \Rightarrow \begin{cases} \mathit{merged}[Y,Z] \approx X \\ \mathit{is\text{–}sorted}[\mathit{merged}[Y,Z]] \end{cases} \right)
\end{array}\right. .$$

The theorem says that,
if

- the predicate ’*is-sorted-version*’ and its sub-operations satisfy the properties described in knowledge (see the appendix),
- the function ’*sorted*’ is defined recursively in the ”divide–and–conquer” style from the auxiliary functions ’*special*’, ’*merged*’, ’*left–split*’, and ’*right–split*’,
- the functions ’*merged*’, ’*left–split*’, and ’*right–split*’ preserve the data type ’*is-tuple*’,
- the functions ’*left–split*’ and ’*right–split*’, on non-trivial arguments, reduce the length,
- the function ’*special*’, on trivial arguments, is the identity,
- the function ’*merged*’, on sorted arguments, yields sorted tuples, and
- the function ’*merged*’, on arguments Y and Z that contain the same elements as *left-split*$[X]$ and *right–split*$[X]$, respectively, yields a tuple that contains the same elements as X,

then

- the function ’*sorted*’ solves the problem of sorting, i.e. the problem specified by the binary predicate ’*is–sorted–version*’.

The most important and most interesting parts of this theorem are the two requirements stating that the function ’*merged*’ preserves sortedness and elements. These two requirements are exactly what people would naturally consider as the characteristic properties of merging. The amazing phenomenon is that exactly these two requirements are invented completely automatically, without any prior intuition or semantic understanding, by our "lazy thinking" method. In fact, the exact formulation of the requirements invented by our method, are slightly more general than the requirements one would expect naturally. This is, of course, good because the weaker the requirements the more functions '*merged*', '*left-split*', and '*right-split*' satisfy the requirements!

Note that our algorithm synthesis method does not only find one particular algorithm for sorting by merging but, rather, finds a whole spectrum of algorithms, namely all those that follow the divide–and–conquer scheme but may have very different instantiations of the subalgorithms '*left–split*', '*right–split*', and '*merged*'. All triples of subalgorithms qualify that satisfy the requirements stated in the relative correctness theorem.

In [5] we go a step further and analyze the knowledge base from which we started with the objective to find out those "minimal" properties of the ingredient operations that suffice for proving the relative correctness theorem. We do not describe this idea in the present paper.

2.6 Mathematical Knowledge Retrieval

After generating the requirements for the sub-functions '*merged*', '*left-split*', and '*right-split*', the question arises whether functions satisfying these requirements already exist in our knowledge base. Seemingly, this is an easy question and, in traditional knowledge retrieval, the question is answered by looking to functions that have these names or, at least, similar names. Thus, for example, if one wants to know what is known about "Bessel functions" in some function library then, of course, one would just look for terms in the library whose outermost function symbol is "Bessel". However, this ad-hoc solution to the knowledge retrieval problem is not appropriate for the needs arising in the frame of the above approach to algorithm synthesis (and in other areas of "mathematical knowledge management").

Rather, we are faced with the following problem:

- Given a knowledge base K, operation names f, $\ldots$, and a requirement on f, $\ldots$, i.e. a formula $R[f,...]$,
- find operation names F, $\ldots$ occurring in K such that $R[F,\ldots]$ is a logical consequence of K.

Hence, knowledge retrieval in our context is essentially a proving problem!

For example, given the knowledge base K in the appendix augmented by the following definitions

$$M[\langle\rangle,\langle\rangle]=\langle\rangle,$$

$$\underset{y,\bar{y}}{\forall}(M[\langle\rangle,\langle y,\bar{y}\rangle]=\langle y,\bar{y}\rangle),$$

$$\underset{x,\bar{x}}{\forall}(M[\langle x,\bar{x}\rangle,\langle\rangle]=\langle x,\bar{x}\rangle),$$

$$\underset{x,\bar{x},y,\bar{y}}{\forall}\left(M[\langle x,\bar{x}\rangle,\langle y,\bar{y}\rangle]=\begin{cases}x\smile M[\langle\bar{x}\rangle,\langle y,\bar{y}\rangle]\Leftarrow x>y\\ y\smile M[\langle x,\bar{x}\rangle,\langle\bar{y}\rangle]\Leftarrow\neg x>y\end{cases}\right),$$

$$\mathrm{L}[\langle\rangle]=\langle\rangle,$$

$$\underset{x}{\forall}(L[\langle x\rangle]=\langle x\rangle),$$

$$\underset{x,y,\bar{z}}{\forall}(L[x,y,\bar{z}]=x\smile L[\langle\bar{z}\rangle]),$$

$$\mathrm{R}[\langle\rangle]=\langle\rangle,$$

$$\underset{x}{\forall}(R[\langle x\rangle]=\langle\rangle),$$

$$\underset{x,y,\bar{z}}{\forall}(R[x,y,\bar{z}]=x\smile R[\langle\bar{z}\rangle])$$

and the following requirement $R[\mathit{left\text{-}split},\mathit{right\text{-}split},\mathit{merged}]$

$$\underset{\substack{is\text{–}tuple[X]\\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall}\begin{cases}is\text{–}tuple[left\text{–}split[X]]\\ X\succ left\text{–}split[X]\\ is\text{–}tuple[right\text{–}split[X]]\\ X\succ right\text{–}split[X]\end{cases},$$

$$\underset{is\text{–}tuple[Y,Z]}{\forall}is\text{–}tuple[merged[Y,Z]],$$

$$\underset{\substack{is\text{–}tuple[X,Y,Z]\\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall}\left(\begin{cases}left\text{–}split[X]\approx Y\\ right\text{–}split[X]\approx Z\\ is\text{–}sorted[Y]\\ is\text{–}sorted[Z]\end{cases}\Rightarrow\begin{cases}merged[Y,Z]\approx X\\ is\text{–}sorted[merged[Y,Z]]\end{cases}\right),$$

then "finding" operations in K that satisfy the requirement consists in trying out all possible triples of functions l,r,m that occur in K and finding out whether the requirement $R[l,r,m]$ can be proved from the formulae in the knowledge base. In our case, in particular, one could try L,R,M and try to prove that $R[L,R,M]$ holds. One sees that this task is nothing else than proving that the algorithms L,R,M are correct w.r.t. to the specification $R[L,R,M]$. Of course, such proofs, may be difficult, moderately difficult, easy, or trivial depending on how much knowledge is available in the knowledge base.

We have explained this in big detail in a recent technical report, see [6]. In particular, if no suitable functions l, r, m are available in the knowledge base, then the problem becomes another synthesis problem, namely the problem of synthesizing algorithms l, r, m satisfying the requirement $R[l, r, m]$. We will solve this problem, again by our lazy thinking algorithm synthesis method in the next section.

3 Second Case Study: Merging by Comparison

3.1 Problem Specification

Now we want to synthesize algorithms 'L', 'R', 'M' that satisfy the specification

$$\underset{\substack{\mathit{is{-}tuple}[X] \\ \neg \mathit{is{-}trivial{-}tuple}[X]}}{\forall} \begin{cases} \mathit{is{-}tuple}[L[X]] \\ X \succ L[X] \\ \mathit{is{-}tuple}[R[X]] \\ X \succ R[X] \end{cases},$$

$$\underset{\mathit{is{-}tuple}[Y,Z]}{\forall} \mathit{is{-}tuple}[M[Y,Z]],$$

$$\underset{\substack{\mathit{is{-}tuple}[X,Y,Z] \\ \neg\text{–trivial–tuple}[X]}}{\forall} \left(\begin{cases} L[X] \approx Y \\ R[X] \approx Z \\ \mathit{is{-}sorted}[Y] \\ \mathit{is{-}sorted}[Z] \end{cases} \Rightarrow \begin{cases} M[Y,Z] \approx X \\ \mathit{is{-}sorted}[M[Y,Z]] \end{cases} \right).$$

In principle, this synthesis would be possible applying again our lazy thinking method. However, it turns out that this synthesis, technically, is very cumbersome. The reason for this is that the above specification of the three unknown algorithms L, R, and M are "coupled". Namely, the last formula in the specification contains all three algorithm names 'L', 'R', and 'M'. In such a situation, it is always advisable trying to "decouple" the specification before one embarks on the synthesis problem. In our case, this is relatively easy: One can prove by pure "symbolic computation proving" (rewriting), i.e. by easy proving without any induction, that the above specification is implied by the following, slightly stronger specification:

$$\underset{\substack{\mathit{is\text{–}tuple}[X] \\ \neg \mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \begin{cases} \mathit{is\text{–}tuple}[L[X]] \\ X \succ L[X] \\ \mathit{is\text{–}tuple}[R[X]] \\ X \succ R[X] \end{cases},$$

$$\underset{\substack{\mathit{is\text{–}tuple}[X] \\ \neg \mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} (L[X] \asymp R[X]) \approx X,$$

$$\underset{\mathit{is\text{–}tuple}[Y,Z]}{\forall} \mathit{is\text{–}tuple}[M[Y,Z]],$$

$$\underset{\mathit{is\text{–}tuple}[Y,Z]}{\forall} \left(\begin{cases} \mathit{is\text{–}sorted}[Y] \\ \mathit{is\text{–}sorted}[Z] \end{cases} \Rightarrow \begin{cases} M[Y,Z] \approx (Y \asymp Z) \\ \mathit{is\text{–}sorted}[M[Y,Z]] \end{cases} \right),$$

where '$\asymp$' denotes concatenation of tuples.

Remark. In fact the second formula is a natural specification for the split functions. What it says is that when we split a tuple X by the functions L and R then, if we put the splits together by concatenation, we again get a tuple that contains exactly the elements of the original X.

Note that, now, the specification of L and R is decoupled from the specification of M. We could now synthesize L and R and then embark on the synthesis of M. In this paper, for lack of space, we do not show the synthesis of some concrete L and R. (In fact, in the previous section, we have provided concrete examples of suitable L and R.) Rather, we only show the synthesis of an algorithm c satisfying

$$\underset{\mathit{is\text{–}tuple}[Y,Z]}{\forall} \mathit{is\text{–}tuple}[c[Y,Z]],$$

$$\underset{\mathit{is\text{–}tuple}[Y,Z]}{\forall} \left(\begin{cases} \mathit{is\text{–}sorted}[Y] \\ \mathit{is\text{–}sorted}[Z] \end{cases} \Rightarrow \begin{cases} c[Y,Z] \approx (Y \asymp Z) \\ \mathit{is\text{–}sorted}[c[Y,Z]] \end{cases} \right).$$

(We are choosing a different name 'c' now in order to indicate that this synthesis problem is completely independent of, i.e. decoupled from, any knowledge on 'L' and 'R'.)

3.2 Algorithm Scheme

We use the following algorithm scheme, which we again suppose to be available in our "algorithm schemes library":

$$c[\langle\rangle, \langle\rangle] = \mathit{cee},$$

$$\underset{y,\bar{y}}{\forall} c[\langle\rangle, \langle y, \bar{y}\rangle] = \mathit{ceg}[y, \bar{y}],$$

$$\underset{x,\bar{x}}{\forall} c[\langle x, \bar{x}\rangle] = \mathit{cge}[x, \bar{x}],$$

$$\underset{x,\bar{x},y,\bar{y}}{\forall} c[\langle x,\bar{x}\rangle,\langle y,\bar{y}\rangle] = \begin{cases} cgg1[x, c[\langle\bar{x}\rangle,\langle y,\bar{y}\rangle]] \Leftarrow p[x,y] \\ cgg2[y, c[\langle x,\bar{x}\rangle,\langle\bar{y}\rangle]] \Leftarrow \neg p[x,y] \end{cases},$$

with the type requirements

$$\begin{aligned} & \textit{is–tuple}[cee], \\ & \underset{x,\bar{x}}{\forall} \textit{is–tuple}[ceg[x,\bar{x}]], \\ & \underset{x,\bar{x}}{\forall} \textit{is–tuple}[cge[x,\bar{x}]], \\ & \underset{x,\textit{is–tuple}[X]}{\forall} \begin{cases} \textit{is–tuple}[cgg1[x,X]] \\ \textit{is–tuple}[cgg2[x,X]] \end{cases}, \end{aligned}$$

where cee, ceg, cge, $cgg1$, $cgg2$, and p are the unknown subalgorithms.

The corresponding inductive proof technique is:
For proving $\underset{\textit{is–tuple}[X]}{\forall} A[X]$ do the following:

(1) Prove $A[\langle\rangle]$.

(2) Take x_0 and $\bar{x_0}$ arbitrary but fixed, assume $A[\langle\bar{x_0}\rangle]$ and prove $A[\langle x_0,\bar{x_0}\rangle]$.

This proof technique can be expanded to the following proof technique for conditioned formulae:
For proving $\underset{\substack{\textit{is–tuple}[X] \\ C[X]}}{\forall} B[X]$ do the following:

(1) Assume $C[\langle\rangle]$and show $B[\langle\rangle]$.

(2) Take x_0 and $\bar{x_0}$ arbitrary but fixed. Assume $\neg C[\langle\bar{x_0}\rangle]$ and $C[\langle x_0,\bar{x_0}\rangle]$and prove $B[\langle x_0,\bar{x_0}\rangle]$.

(3) Now assume $C[\langle\bar{x_0}\rangle]$, $B[\langle\bar{x_0}\rangle]$ and $C[\langle x_0,\bar{x_0}\rangle]$ and prove $B[\langle x_0,\bar{x_0}\rangle]$.

Note that (2) and (3) are appropriate because the assumption $C[\langle\bar{x_0}\rangle] \Rightarrow B[\langle\bar{x_0}\rangle]$ is equivalent to $\neg C[\langle\bar{x_0}\rangle] \vee (C[\langle\bar{x_0}\rangle] \wedge B[\langle\bar{x_0}\rangle])$ and, hence, the proof splits into the two cases $\neg C[\langle\bar{x_0}\rangle]$ and $(C[\langle\bar{x_0}\rangle] \wedge B[\langle\bar{x_0}\rangle])$.

3.3 Algorithm Synthesis by Lazy Thinking: First Round

We first prove

$$\underset{\textit{is–tuple}[Y,Z]}{\forall} \textit{is–tuple}[c[Y,Z]]$$

by induction on Y and Z.

This easy proof succeeds because of the type requirements on the subalgorithms cee, ceg, etc. and the algorithm scheme. Thus, we immediately continue with the proof.

Now we try to prove

$$\underset{is\text{–}tuple[Y,Z]}{\forall}\left(\begin{cases} is\text{–}sorted[Y] \\ is\text{–}sorted[Z] \end{cases} \Rightarrow \begin{cases} c[Y,Z] \approx (Y \asymp Z) \\ is\text{–}sorted[c[Y,Z]] \end{cases}\right)$$

by induction for formulae with conditions.

Induction Base for Y:

Assume $is\text{–}sorted[\langle\rangle]$. Prove

$$\underset{\substack{is\text{–}tuple[Z] \\ is\text{–}sorted[Z]}}{\forall}\begin{cases} \langle\rangle \asymp Z \approx c[\langle\rangle, Z] \\ is\text{–}sorted[c[\langle\rangle, Z]] \end{cases}.$$

Induction Base for Y, Induction Base for Z:

Assume $is\text{–}sorted[\langle\rangle]$. Prove

$$\langle\rangle \asymp \langle\rangle \approx c[\langle\rangle, \langle\rangle],$$
$$is\text{–}sorted[\mathrm{c}[\langle\rangle, \langle\rangle]].$$

Now, $\langle\rangle \asymp \langle\rangle \approx c[\langle\rangle, \langle\rangle] \leftrightarrow \langle\rangle \approx cee$.

This proof fails but straightforwardly leads to the requirement

$$cee = \langle\rangle.$$

Furthermore, using already this requirement,

$$is\text{–}sorted[c[\langle\rangle, \langle\rangle]] \leftrightarrow is\text{–}sorted[cee] \leftrightarrow is\text{–}sorted[\langle\rangle] \leftrightarrow \text{true}.$$

Induction Base for Y, Induction Step for Z, First Case:

Assume $\neg is\text{–}sorted[\langle\bar{z}_0\rangle]$, $is\text{–}sorted[\langle z_0, \bar{z}_0\rangle]$.

Prove $\langle\rangle \asymp \langle z_0, \bar{z}_0\rangle \approx c[\langle\rangle, \langle z_0, \bar{z}_0\rangle]$ and $is\text{–}sorted[c[\langle\rangle, \langle z_0, \bar{z}_0\rangle]]$.

The assumption $\neg is\text{–}sorted[\langle\bar{z}_0\rangle]$ and $is\text{–}sorted[\langle z_0, \bar{z}_0\rangle]$ contradict each other. Hence, this case is not possible.

Induction Base for Y, Induction Step for Z, Second Case:

Assume

$is\text{–}sorted[\langle\bar{z}_0\rangle]$, $\langle\rangle \asymp \langle\bar{z}_0\rangle \approx c[\langle\rangle, \langle\bar{z}_0\rangle]$, $is\text{–}sorted[c[\langle\rangle, \langle\bar{z}_0\rangle]]$, $is\text{–}sorted[\langle z_0, \bar{z}_0\rangle]$.

Prove $\langle\rangle \asymp \langle z_0, \bar{z}_0\rangle \approx c[\langle\rangle, \langle z_0, \bar{z}_0\rangle]$ and $is\text{–}sorted[c[\langle\rangle, \langle z_0, \bar{z}_0\rangle]$.

CASE $\bar{z}_0$ =(the empty sequence):

$$\langle\rangle \asymp \langle z_0, \bar{z}_0\rangle \approx c[\langle\rangle, \langle z_0, \bar{z}_0\rangle] \leftrightarrow \langle z_0\rangle \approx ceg[z_0] \leftrightarrow (\langle z_0\rangle \in ceg[z_0] \wedge \langle\rangle \approx dfo[z_0, ceg[z_0]]) \leftarrow (z_0 \in ceg[z_0] \wedge (dfo[z_0, ceg[z_0]] = \langle\rangle)).$$

This proof fails but (by our requirement generation algorithm automatically) leads to the requirement

$$\underset{z}{\forall}(z \in ceg[z] \wedge (dfo[z, ceg[z] = \langle\rangle)).$$

Furthermore,

$$is\text{–}sorted[c[\langle\rangle, \langle z_0, \bar{z}_0\rangle] \leftrightarrow is\text{–}sorted[c[\langle\rangle, \langle z_0\rangle]] \leftrightarrow is\text{–}sorted[ceg[z_0]].$$

The proof fails but (automatically) leads to the requirement

$$\underset{z}{\forall}(\textit{is–sorted}[ceg[z]]).$$

3.4 Algorithm Synthesis by Lazy Thinking: Second Round

It is clear that, with the requirements for the subalgorithm '*ceg*' introduced in the preceding round, one can get over the proof situation in which the first proof attempt was stuck and we come to the following proof situation:

Induction Base for Y, Induction Step for Z, Second Case:

Assume
$\textit{is–sorted}[\langle\bar{z_0}\rangle]$, $\langle\rangle \asymp \langle\bar{z_0}\rangle \approx c[\langle\rangle, \langle\bar{z_0}\rangle]$, $\textit{is–sorted}[c[\langle\rangle, \langle\bar{z_0}\rangle]]$, $\textit{is–sorted}[\langle z_0, \bar{z_0}\rangle]$.
Prove $\langle\rangle \asymp \langle z_0, \bar{z_0}\rangle \approx c[\langle\rangle, \langle z_0, \bar{z_0}\rangle]$ and $\textit{is–sorted}[c[\langle\rangle, \langle z_0, \bar{z_0}\rangle]]$.

CASE $\bar{z_0} = z_1, \bar{z}$ (for certain $z_1, \bar{z}$):
From the assumptions we know:

$$\langle\rangle \asymp \langle z_1, \bar{z}\rangle \ \approx c[\langle\rangle, \langle z_1, \bar{z}\rangle] \leftrightarrow \langle z_1, \bar{z}\rangle \approx ceg[z_1, \bar{z}],$$

$$\textit{is–sorted}[c[\langle\rangle, \langle\bar{z_0}\rangle]] \leftrightarrow \textit{is–sorted}[ceg[z_1, \bar{z}]],$$

$$(\textit{is–sorted}[\langle z_0, \bar{z_0}\rangle] \leftrightarrow \textit{is–sorted}[\langle z_0, z_1, \bar{z}\rangle]$$

$$\leftrightarrow (z_0 \geq z_1 \wedge \textit{is–sorted}[\langle z_1, \bar{z}\rangle]) \leftrightarrow z_0 \geq z_1 \ .$$

Now,
$\langle\rangle \asymp \langle z_0, \bar{z_0}\rangle \approx c[\langle\rangle, \langle z_0, \bar{z_0}\rangle] \leftrightarrow \langle z_0, z_1, \bar{z}\rangle \approx ceg[z_0, z_1, \bar{z}]$
$\leftrightarrow (z_0 \in ceg[z_0, z_1, \bar{z}] \wedge \langle z_1, \bar{z}\rangle \approx dfo[z_0, ceg[z_0, z_1, \bar{z}]])$
$\leftarrow (z_0 \in ceg[z_0, z_1, \bar{z}] \wedge (ceg[z_1, \bar{z}] \approx dfo[z_0, ceg[z_0, z_1, \bar{z}]]))$.
This proof fails but it leads to the requirement

$$\underset{z_0, z_1, \bar{z}}{\forall}(z_0 \in ceg[z_0, z_1, \bar{z}] \wedge (ceg[z_1, \bar{z}] \approx dfo[z_0, ceg[z_0, z_1, \bar{z}]])).$$

Furthermore,

$$\textit{is–sorted}[c[\langle\rangle, \langle z_0, \bar{z_0}\rangle]] \leftrightarrow \textit{is–sorted}[c[\langle\rangle, \langle z_0, z_1, \bar{z}\rangle]] \leftrightarrow \textit{is–sorted}[ceg[z_0, z_1, \bar{z}]].$$

The proof fails, but it leads to the requirement

$$\underset{z_0, z_1, \bar{z}}{\forall}\left(\left\{\begin{array}{l} \textit{is–sorted}[ceg[z_1, \bar{z}]] \\ z_0 \geq z_1 \end{array}\right. \Rightarrow \textit{is–sorted}[ceg[z_0, z_1, \bar{z}]]\right).$$

3.5 Algorithm Synthesis by Lazy Thinking: Third Round

It is clear that, with the additional requirements for the subalgorithm '*ceg*' introduced in the preceding round, one can get over the proof situation in which the proof attempt in the preceding round was stuck and we come to the following proof situation

Induction Step for Y, First Case:

Assume $\neg$*is–sorted*$[\langle \bar{y_0}\rangle]$ and *is–sorted*$[\langle y_0, \bar{y_0}\rangle]$. The assumptions contradict each other and, hence, this case is not possible.

Induction Step for Y, Second Case, Induction Base for Z:

Assume:

$$\begin{aligned}
&\mathit{is\text{–}sorted}[\langle \bar{y_0}\rangle],\\
&\begin{cases} \langle \bar{y_0}\rangle \asymp \langle\rangle \approx c[\langle \bar{y_0}\rangle, \langle\rangle] \\ \mathit{is\text{–}sorted}[c[\langle \bar{y_0}\rangle, \langle\rangle]] \end{cases},\\
&\mathit{is\text{–}sorted}[\langle y_0, \bar{y_0}\rangle]\ .
\end{aligned}$$

Prove

$$\begin{cases} \langle y_0, \bar{y_0}\rangle \asymp \langle\rangle \approx c[\langle y_0, \bar{y_0}\rangle, \langle\rangle] \\ \mathit{is\text{–}sorted}[c[\langle y_0, \bar{y_0}\rangle, \langle\rangle]] \end{cases}.$$

We will skip the details of the proof here. In fact it is similar to the proof in the previous round of exploration. The failing situations in the proof will lead to the following requirements:

$$\begin{aligned}
&\underset{y}{\forall}\, \mathit{is\text{–}sorted}[\mathit{cge}[y]],\\
&\underset{y_0,y_1,\bar{y}}{\forall} (y_0 \in \mathit{cge}[y_0, y_1, \bar{y}] \wedge (\mathit{cge}[y_1, \bar{y}] \approx \mathit{dfo}[y_0, \mathit{cge}[y_0, y_1, \bar{y}]])),\\
&\underset{y_0,y_1,\bar{y}}{\forall} \left(\begin{cases} \mathit{is\text{–}sorted}[\mathit{cge}[y_1, \bar{y}]] \\ y_0 \geq y_1 \end{cases} \Rightarrow \mathit{is\text{–}sorted}[\mathit{cge}[y_0, y_1, \bar{y}]] \right)\ .
\end{aligned}$$

3.6 Algorithm Synthesis by Lazy Thinking: Fourth Round

It is clear that, with the additional requirements for the subalgorithm '*cge*' introduced in the preceding round, one can get over the proof situation in which the proof attempt in the third round was stuck and we come to the following proof situation:

Induction Step for Y, Second Case, Induction Step for Z, First Case:

Assume $\neg$*is–sorted*$[\langle \bar{z_0}\rangle]$ and *is–sorted*$[\langle z_0, \bar{z_0}\rangle]$. The assumptions contradict each other. Hence, this case is not possible.

Induction Step for Y, Second Case, Induction Step for Z, Second Case:

Assume

$$\begin{aligned}
&\textit{is–sorted}[\langle \bar{z_0}\rangle],\\
&\langle y_0, \bar{y_0}\rangle \asymp \langle \bar{z_0}\rangle \approx c[\langle y_0, \bar{y_0}\rangle, \langle \bar{z_0}\rangle] \leftrightarrow \langle y_0, \bar{y_0}, \bar{z_0}\rangle \approx c[\langle y_0, \bar{y_0}\rangle, \langle \bar{z_0}\rangle],\\
&\textit{is–sorted}[c[\langle y_0, \bar{y_0}\rangle, \langle \bar{z_0}\rangle]],\\
&\textit{is–sorted}[\langle z_0, \bar{z_0}\rangle]\ .
\end{aligned}$$

Prove

$$\langle y_0, \bar{y_0}\rangle \asymp \langle z_0, \bar{z_0}\rangle \approx c[\langle y_0, \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle] \leftrightarrow \langle y_0, \bar{y_0}, z_0, \bar{z_0}\rangle \approx c[\langle y_0, \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]$$

and

$$\textit{is–sorted}[c[\langle y_0, \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]].$$

Now we distinguish two cases.

CASE $p[y_0, z_0]$:

In this case,

$\langle y_0, \bar{y_0}, z_0, \bar{z_0}\rangle \approx c[\langle y_0, \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle] \leftrightarrow \langle y_0, \bar{y_0}, z_0, \bar{z_0}\rangle \approx cgg1[y_0, c[\langle \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]]$ $(\star)$

By the induction hypothesis on Y

$$\mathop{\forall}_{\substack{\textit{is–tuple}[Z]\\ \textit{is–sorted}[Z]}} \begin{cases} \langle \bar{y_0}\rangle \asymp Z \approx c[\langle \bar{y_0}\rangle, Z] \\ \textit{is–sorted}[c[\langle \bar{y_0}\rangle, Z]] \end{cases}$$

we know, in particular,

$$\langle \bar{y_0}, z_0, \bar{z_0}\rangle \approx c[\langle \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle].$$

We are now stuck at the proof of $(\star)$.

Now, with the requirements

$$\mathop{\forall}_{x,\textit{is–tuple[Y, Z]}} (Y \approx Z \Rightarrow cgg1[x, Y] \approx cgg1[x, Z]),$$
$$\mathop{\forall}_{y,\bar{y},z,\bar{z}} \langle y, \bar{y}, z, \bar{z}\rangle \approx cgg1[y, \langle \bar{y}, z, \bar{z}\rangle]$$

we can prove $(\star)$:

$$(\star) \leftrightarrow \langle y_0, \bar{y_0}, z_0, \bar{z_0}\rangle \approx cgg1[y_0, \langle \bar{y_0}, z_0, \bar{z_0}\rangle] \leftrightarrow \text{true}\ .$$

Furthermore,

$\textit{is–sorted}[c[\langle y_0, \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]] \leftrightarrow \textit{is–sorted}[cgg1[y_0, c[\langle \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]]]$ $(\star\star)$.

We already know, by induction hypothesis on Y:

$$\begin{aligned}
&\textit{is–sorted}[c[\langle \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]],\\
&\langle \bar{y_0}, z_0, \bar{z_0}\rangle \approx c[\langle \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]
\end{aligned}$$

and we also know

$$\textit{is–sorted}[\langle y_0, \bar{y_0}\rangle].$$

We are now stuck at the proof of $(\star\star)$.

However, with the (automatically generated) requirement

$$\mathop{\forall}\limits_{y,\bar{y},z,\bar{z},\mathit{is\text{--}tuple}[Y]}\left(\left\{\begin{array}{l}\mathit{is\text{--}sorted}[Y]\\ Y\approx\langle\bar{y},z,\bar{z}\rangle\\ \mathit{is\text{--}sorted}[\langle y,\bar{y}\rangle]\\ \mathit{is\text{--}sorted}[\langle z,\bar{z}\rangle]\\ p[y,z]\end{array}\right.\Rightarrow \mathit{is\text{--}sorted}[\mathit{cgg}1[y,Y]]\right)$$

we can successfully get over this proof situation.

CASE $\neg p[y_0, z_0]$:

This case is similar. We do not give the details. From the proof situation in which the proof in this branch is stuck, the following requirements can be generated (automatically):

$$\mathop{\forall}\limits_{x,\mathit{is\text{--}tuple}[Y,\ Z]}(Y\approx Z\Rightarrow \mathit{cgg}2[x,Y]\approx \mathit{cgg}2[x,Z]),$$

$$\mathop{\forall}\limits_{y,\bar{y},z,\bar{z}}\langle y,\bar{y},z,\bar{z}\rangle\approx \mathit{cgg}2[z,\langle y,\bar{y},\bar{z}\rangle],$$

$$\mathop{\forall}\limits_{y,\bar{y},z,\bar{z},\mathit{is\text{--}tuple}[Y]}\left(\left\{\begin{array}{l}\mathit{is\text{--}sorted}[Y]\\ Y\approx\langle y,\bar{y},\bar{z}\rangle\\ \mathit{is\text{--}sorted}[\langle y,\bar{y}\rangle]\\ \mathit{is\text{--}sorted}[\langle z,\bar{z}\rangle]\\ \neg p[y,z]\end{array}\right.\Rightarrow \mathit{is\text{--}sorted}[\mathit{cgg}2[z,Y]]\right).$$

3.7 Summary of the Requirements for the Subalgorithms

We now collect all the requirements on the subalgorithms *cee*, *ceg*, *cge*, *cgg*1, *cgg*2:

$\mathit{is\text{--}tuple}[\mathit{cee}]$,

$\mathop{\forall}\limits_{x,\bar{x}}\mathit{is\text{--}tuple}[\mathit{ceg}[x,\bar{x}]]$,

$\mathop{\forall}\limits_{x,\bar{x}}\mathit{is\text{--}tuple}[\mathit{cge}[x,\bar{x}]]$,

$$\mathop{\forall}\limits_{x,\mathit{is\text{--}tuple}[X]}\left\{\begin{array}{l}\mathit{is\text{--}tuple}[\mathit{cgg}1[x,X]]\\ \mathit{is\text{--}tuple}[\mathit{cgg}2[x,X]]\end{array}\right.,$$

cee= $\langle\rangle$,

$\mathop{\forall}\limits_{z}(z\in \mathit{ceg}[z]\wedge(\mathit{dfo}[z,\mathit{ceg}[z]])=\langle\rangle$,

$$\underset{z}{\forall}(\mathit{is\text{--}sorted}[ceg[z]]),$$

$$\underset{z_0,z_1,\bar{z}}{\forall}(z_0 \in cge[z_0,z_1,\bar{z}] \wedge (ceg[z_1,\bar{z}] \approx dfo[z_0, ceg[z_0,z_1,\bar{z}]])),$$

$$\underset{z_0,z_1,\bar{z}}{\forall}\left(\left\{\begin{array}{l} \mathit{is\text{--}sorted}[ceg[z_1,\bar{z}]] \\ z_0 \geq z_1 \end{array}\right. \Rightarrow \mathit{is\text{--}sorted}[ceg[z_0,z_1,\bar{z}]]\right),$$

$$\underset{y}{\forall}\, \mathit{is\text{--}sorted}[cge[y]],$$

$$\underset{y_0,y_1,\bar{y}}{\forall}(y_0 \in cge[y_0,y_1,\bar{y}] \wedge (cge[y_1,\bar{y}] \approx dfo[y_0, cge[y_0,y_1,\bar{y}]])),$$

$$\underset{y_0,y_1,\bar{y}}{\forall}\left(\left\{\begin{array}{l} \mathit{is\text{--}sorted}[cge[y_1,\bar{y}]] \\ y_0 \geq y_1 \end{array}\right. \Rightarrow \mathit{is\text{--}sorted}[cge[y_0,y_1,\bar{y}]]\right),$$

$$\underset{x,\mathit{is\text{--}tuple[Y,\ Z]}}{\forall}(Y \approx Z \Rightarrow cgg1[x,Y] \approx cgg1[x,Z]),$$

$$\underset{y,\bar{y},z,\bar{z}}{\forall}\langle y,\bar{y},z,\bar{z}\rangle \approx cgg1[y,\langle \bar{y},z,\bar{z}\rangle],$$

$$\underset{y,\bar{y},z,\bar{z},\mathit{is\text{--}tuple}[Y]}{\forall}\left(\left\{\begin{array}{l} \mathit{is\text{--}sorted}[Y] \\ Y \approx \langle \bar{y},z,\bar{z}\rangle \\ \mathit{is\text{--}sorted}[\langle y,\bar{y}\rangle] \\ \mathit{is\text{--}sorted}[\langle z,\bar{z}\rangle] \\ p[y,z] \end{array}\right. \Rightarrow \mathit{is\text{--}sorted}[cgg1[y,Y]]\right),$$

$$\underset{x,\mathit{is\text{--}tuple[Y,\ Z]}}{\forall}(Y \approx Z \Rightarrow cgg2[x,Y] \approx cgg2[x,Z]),$$

$$\underset{y,\bar{y},z,\bar{z}}{\forall}\langle y,\bar{y},z,\bar{z}\rangle \approx cgg2[z,\langle y,\bar{y},\bar{z}\rangle],$$

$$\underset{y,\bar{y},z,\bar{z},\mathit{is\text{--}tuple}[Y]}{\forall}\left(\left\{\begin{array}{l} \mathit{is\text{--}sorted}[Y] \\ Y \approx \langle y,\bar{y},\bar{z}\rangle \\ \mathit{is\text{--}sorted}[\langle y,\bar{y}\rangle] \\ \mathit{is\text{--}sorted}[\langle z,\bar{z}\rangle] \\ \neg p[y,z] \end{array}\right. \Rightarrow \mathit{is\text{--}sorted}[cgg2[z,Y]]\right).$$

The most natural definitions of algorithms satisfying these requirements are the following:

$$cee = \langle\rangle,$$

$$\underset{\bar{x}}{\forall}(ceg[\bar{x}] = \langle \bar{x}\rangle),$$

$$\underset{\bar{y}}{\forall}(cge[\bar{y}] = \langle\bar{y}\rangle),$$

$$\underset{x,is\text{–}tuple[Z]}{\forall}(cgg1[x, Z] = x \smile Z),$$

$$\underset{y,is\text{–}tuple[Z]}{\forall}(cgg2[y, Z] = y \smile Z)\ .$$

(However, if one embarks on synthesizing *cgg*1, *cgg*2 from the elementary tuple operations available in the knowledge base, it is again advisable to first simplify the above requirements to a few essential ones using simple symbolic computation proof techniques, without induction.)

4 Third Case Study: Sorting by Insertion

4.1 Remark

Of course, sorting by insertion can be viewed as a degenerate case of sorting by merging: If *left-split* degenerates to the function that just takes the left-most element out of a given tuple then sorting by merging (an $n \log n$ algorithm) becomes sorting by insertion (an n^2 algorithm).

The interest of the third case study does, therefore, not lie in the fact that sorting by insertion can be synthesized but lies in the investigation of what happens if we start with the same algorithm specification of '*is-sorted-version*' as in the first case study, namely,

$$\underset{is\text{–}tuple[X]}{\forall} is\text{–}sorted\text{–}version[X, sorted[X]]$$

but throw in a different algorithm scheme.

4.2 Algorithm Scheme

This time we use the following algorithm scheme

$$sorted[\langle\rangle] = c,$$

$$\underset{x}{\forall}(sorted[\langle x\rangle] = d[x]),$$

$$\underset{x,y,\bar{z}}{\forall}(sorted[\langle x, y, \bar{z}\rangle] = i[x, sorted\langle y, \bar{z}\rangle]),$$

with the following natural type requirements on the auxiliary functions

$$is\text{–}tuple[c],$$

$$\underset{x}{\forall} is\text{–}tuple[d[x]],$$

$$\underset{x,is\text{–}tuple[Y]}{\forall} is\text{–}tuple[i[x, Y]]\ .$$

The corresponding inductive proof technique is:

For proving $\underset{is\text{–}tuple[X]}{\forall} A[X]$ do the following:

(1) Prove $A[\langle\rangle]$.

(2) Take x_0 arbitrary but fixed and prove $A[\langle x_0\rangle]$.

(3) Take x_0, y_0 and $\bar{z}_0$ arbitrary but fixed, assume $A[\langle y_0, \bar{z}_0\rangle]$ and prove $A[\langle x_0, y_0, \bar{z}_0\rangle]$.

4.3 Algorithm Synthesis

We start with an attempt to prove

$$\underset{is\text{–}tuple[X]}{\forall} is\text{–}sorted\text{–}version[X, sorted[X]].$$

We do not show the proof attempts for the two base cases. Analyzing the failing situations in these two attempts readily leads to the two requirements

$$c = \langle\rangle$$

and

$$\underset{x}{\forall} d[x] = \langle x\rangle.$$

Now we do the induction step. For this, we take x_0, y_0, and $\bar{z}_0$ arbitrary but fixed and assume

$$is\text{–}sorted\text{–}version[\langle y_0, \bar{z}_0\rangle, sorted[\langle y_0, \bar{z}_0\rangle]]$$

i.e.

$$\begin{gathered} is\text{–}tuple[sorted[\langle y_0, \bar{z}_0\rangle]] \;\; (A1), \\ \langle y_0, \bar{z}_0\rangle \approx sorted[\langle y_0, \bar{z}_0\rangle] \; (A2), \\ is\text{–}sorted[sorted[\langle y_0, \bar{z}_0\rangle]] \; (A3), \end{gathered}$$

and prove

$$is\text{–}sorted\text{–}version[\langle x_0, y_0, \bar{z}_0\rangle, sorted[\langle x_0, y_0, \bar{z}_0\rangle]]$$

i.e

$$is\text{–}sorted\text{–}version[\langle x_0, y_0, \bar{z}_0\rangle, i[x_0, sorted[\langle y_0, \bar{z}_0\rangle]]]$$

which we expand to

$$\begin{gathered} is\text{–}tuple[i[x_0, sorted[\langle y_0, \bar{z}_0\rangle]]] \quad (G1), \\ \langle x_0, y_0, \bar{z}_0\rangle \approx i[x_0, sorted[\langle y_0, \bar{z}_0\rangle]] \; (G2), \\ is\text{–}sorted[i[x_0, sorted[\langle y_0, \bar{z}_0\rangle]]] \quad (G3). \end{gathered}$$

Now, $(G1)$ follows from the type requirement on i and $(A1)$.

The proof of $(G2)$ and $(G3)$ fails. However, the following requirements for i can be readily (i.e. automatically) generated from the temporary assumptions and goals:

$$\underset{x,y,\bar{z},is\text{–}tuple[Y]}{\forall} \left(\left\{ \begin{array}{l} \langle y, \bar{z}\rangle \approx Y \\ is\text{–}sorted[Y] \end{array} \right. \Rightarrow \left\{ \begin{array}{l} \langle x, y, \bar{z}\rangle \approx i[x, Y] \\ is\text{–}sorted[i[x, Y]] \end{array} \right. \right).$$

With these requirements, the proof can be completed.

In fact, together with the type requirement on i, these requirements characterize the natural properties of "insertion algorithms".

5 Automation of the Lazy Thinking Algorithm Synthesis Procedure

The "lazy thinking" procedure for inventing algorithms together with their correctness proofs can be made completely automatic (algorithmic) if we manage

- to automate (inductive) proving in the specific area and
- to automate generating conjectures (requirements on auxiliary functions) from the temporary assumptions and the left-over goals in failing proof attempts.

In fact, for the case of inductive domains, there are powerful automated provers around and we have implemented various such provers in the *Theorema* system. Also, we already implemented a first version of a conjecture generation algorithm which, together with our automated inductive provers, is powerful enough to completely automate the "lazy thinking" algorithm invention and verification process in the case of quite some problems on tuples.

Our current conjecture (requirements) generation algorithm implements two stratregies that can handle the two situations in the above example but also in many other examples. Both strategies take the conjunction A of all temporary assumptions and the (conjunction of the) temporary goals G and conjecture a variant of $(A \Rightarrow G)$:

- The first strategy can handle simple failing proof situations in proofs (proof branches) without induction: It replaces all "arbitrary but fixed" constants in $(A \Rightarrow G)$ by variables v,... and produces the conjecture $\underset{v,...}{\forall}(A \Rightarrow G)$. By this strategy, one can produce, for example, the conjecture (requirement)
$$\underset{is\text{–}trivial\text{–}tuple[X]}{\forall}(special[X] = X)$$
in the first round of the first case study.
- The second strategy can handle failing proof situations in the induction step parts of proofs. It first looks in $(A \Rightarrow G)$ for terms whose head is the function constant for the algorithm to be synthesized (in our case, this is the function constant '*sorted*') and replaces these with new variables v,... of the same type as the output of the unknown function and then, again, replaces all "arbitrary but fixed" constants by variables w,.... The variant $\underset{v,...,w,...}{\forall}(A \Rightarrow G)$ is taken as the new conjecture. By this strategy, one can produce the requirements in the induction steps of all the above case studies, for example the requirement

$$\underset{\substack{is\text{–}tuple[X,Y,Z]\\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall}\left(\begin{cases} left\text{–}split[X] \approx Y \\ right\text{–}split[X] \approx Z \\ is\text{–}sorted[Y] \\ is\text{–}sorted[Z] \end{cases} \Rightarrow \begin{cases} merged[Y,Z] \approx X \\ is\text{–}sorted[merged[Y,Z]] \end{cases}\right)$$

in the first case study.

Our future research will focus on adding more and more strategies to the conjecture generation algorithm. Of course, never, one conjecture generation algorithm will be able to handle "all" failing proof situations. However, we think that the lazy thinking cascade will be a useful tool for organizing the theory exploration process and, in particular, the algorithm invention process. The cascade becomes more and more powerful the more powerful theorem provers and conjecture generation algorithms will be used as subalgorithms and the better we understand and organize libraries of algorithm schemes.

With the current *Theorema* induction prover and the current *Theorema* conjecture generator, the above synthesis process can be executed completely automatically. This means that the user has only to compile the knowledge on the predicate '*is-sorted-version*' and its auxiliary notions shown in the appendix and then to call *Theorema* by

$$\begin{aligned}&Prove[Theorem[\text{"correctness of sorting"}],\\ &using \to Theory[\text{"sorting"}],\\ &by \to Cascade[SqnsEqCasePC, GenerateConjectures]]\end{aligned}$$

Here, $Theory[\text{"sorting"}]$ is the name of the theory consisting of the formulae in the appendix. In *Theorema*, this name can be assigned to the formulae by executing

Theory["sorting",

$$\underset{is\text{–}tuple[X]}{\forall}\left(is\text{–}sorted\text{–}version[X,Y] \Leftrightarrow \begin{cases} is\text{–}tuple[Y] \\ X \approx Y \\ is\text{–}sorted[Y] \end{cases}\right)$$

$$is\text{–}sorted[\langle\rangle]$$

$$\underset{x}{\forall} is\text{–}sorted[\langle x\rangle]$$

... all formulae in the appendix ...]

Similarly, $Theorem[\text{"correctness of sorting"}]$ is the name of the correctness theorem for sorting. This name can be assigned by executing

Theorem["correctness of sorting",

$$\underset{is\text{–}tuple[X]}{\forall} is\text{–}sorted\text{–}version[X, sorted[X]]]$$

'*SqnsEqCasePC*' is the name of the particular induction prover that corresponds to the "divide–and–conquer" algorithm scheme. This prover adds the formulae that constitute the algorithm scheme, i.e. the formulae

$$\underset{\mathit{is\text{–}tuple}[X]}{\forall} \mathit{sorted}[X] = \begin{cases} \mathit{special}[X] & \Leftarrow \mathit{is\text{–}trivial\text{–}tuple}[X] \\ \mathit{merged}[\mathit{sorted}[\mathit{left\text{–}split}[X]], & \\ \qquad \mathit{sorted}[\mathit{right\text{–}split}[X]]] & \text{otherwise} \end{cases},$$

$$\underset{\substack{\mathit{is\text{–}tuple}[X] \\ \neg \mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \quad \mathit{is\text{–}tuple}[\mathit{left\text{–}split}[X]],$$

$\ldots etc \ldots$

to the knowledge and organizes the main loop of the proof by the particular induction scheme. The strategy implemented by this prover is the following: first apply the induction scheme, then expand the case statement (from the definition of the algorithm), then finally apply rewriting and natural deduction to carry the proof through.

We are now working on a general induction prover that gets the information on the algorithm scheme (including the type requirements for the auxiliary functions and the appropriate induction scheme) directly from the library of algorithm schemes so that, without user interaction in between, the prover can attempt various algorithm syntheses successively.

Future work will also be dedicated to the development of algorithm type libraries and classification of algorithm types (some of this type of work has already been carried out by D.R. Smith, for instance in [18]).

As result of the above *Theorema* call '*Prove*[*Theorem*["correctness of sorting"],...]', after approximately 5 minutes computation time (on a Compaq Evo N610c, with Intel Pentium 4 1.8 GHz), the user will get

- an augmented knowledge base that contains the requirements on the auxiliary functions

$$\underset{\mathit{is\text{–}trivial\text{–}tuple}[X]}{\forall} (\mathit{special}[X] = X),$$

$$\underset{\substack{\mathit{is\text{–}tuple}[X,Y,Z] \\ \neg \mathit{is\text{–}trivial\text{–}tuple}[X]}}{\forall} \left(\begin{cases} \mathit{left\text{–}split}[X] \approx Y \\ \mathit{right\text{–}split}[X] \approx Z \\ \mathit{is\text{–}sorted}[Y] \\ \mathit{is\text{–}sorted}[Z] \end{cases} \Rightarrow \begin{cases} \mathit{merged}[Y,Z] \approx X \\ \mathit{is\text{–}sorted}[\mathit{merged}[Y,Z]] \end{cases} \right)$$

- and a complete correctness proof for the divide–and–conquer algorithm that essentially looks like the proof developed in the preceding sections.

6 Conclusion

We presented a procedure for automated algorithm invention and verification. The proposed procedure

- is natural
- can also be used as a heuristic and didactic guide for the development of correct algorithms and their correctness proofs
- uses algorithm schemes as condensed algorithmic knowledge
- exploits the information gained from failing proof attempts of the correctness theorem
- is able to generate conjectures (requirements on the sub-algorithms) from failing proofs
- invents verified algorithms that can be used with an infinite spectrum of possible subalgorithms (all those that satisfy the requirements)
- emphasizes a layered approach in repeated, small extensions of theories.

References

[1] Armando, A., A.Smaill, I. Green, *Automatic Synthesis of Recursive Programs: the Proof-Planning Paradigm.* Automated Software Engineering, **Vol. 6**(1999), pp. 329-356 .

[2] Asperti, A. , B. Buchberger, J. H. Davenport (eds), "Mathematical Knowledge Management, Proc. of the Second International Conference on Mathematical Knowledge Management, MKM 2003, February 2003, Bertinoro, Italy". Lecture Notes in Computer Science, **Vol. 2594**(2003), Springer, Berlin - Heidelberg - New York, 223 pages.

[3] Basin, D. , Y. Deville, P. Flener, A. Hamfelt, J. Fischer Nilsson. *Synthesis of Programs in Computational Logic.* In: M. Bruynooghe and K. K. Lau, "Program Development in Computational Logic", Springer-Verlag, to appear.

[4] Buchberger, B., *Theory Exploration with Theorema.* Analele Universitatii Din Timisoara, Ser. Matematica-Informatica, **Vol. XXXVIII**, Fasc.2, (2000), ("Proceedings of SYNASC 2000, 2nd International Workshop on Symbolic and Numeric Algorithms in Scientific Computing", Oct. 4-6, 2000, Timisoara, Rumania, T. Jebelean, V. Negru, A. Popovici eds.), pp. 9-32.

[5] Buchberger, B., *Algorithm Invention and Verification by Lazy Thinking.* In: D. Petcu, V. Negru, D. Zaharie, T. Jebelean (eds), "Proceedings of SYNASC 2003 (Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, October 1-4, 2003)", Mirton Publishing, ISBN 973-661-104-3, pp. 2-26.

[6] Buchberger, B., *Algorithm Retrieval: Concept Clarification and Case Study in Theorema.* SFB Scientific Computing, Technical Report 2003-25, Research

Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, October 2003.

[7] Buchberger,B., O. Caprotti (eds.), "Electronic Proceedings of the 1st International Workshop of Mathematical Knowledge Management", September 24-26, 2001, Research Institute for Symbolic Computation (RISC), Hagenberg, Austria, http://www.risc.uni-linz.ac.at/institute/conferences/MKM2001/Proceedings/.

[8] Buchberger, B., T. Jebelean, F. Kriftner, M. Marin, D. Vasaru, *An Overview on the Theorema Project*, In: W. Kuechlin (ed.), "Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997)", ACM Press 1997, pp. 384-391.

[9] Buchberger, B., C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger, *Theorema: A Progress Report*. In: M. Kerber and M. Kohlhase (eds.), "Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, August 6-7, 2000, St. Andrews, Scotland)", A.K. Peters, Natick, Massachusetts, pp. 98-113.

[10] Buchberger, B., G. Gonnet, M. Hazewinkel (eds.) , "Mathematical Knowledgement Management", Special Issue of the Journal Annals of Mathematics and Artificial Intelligence, **Vol. 38**(2003), Nos. 1-3, Kluwer Academic Publishers.

[11] Farmer, W.(ed.), The First North American Workshop on MKM (NA-MKM 2002) in Hamilton, Ontario, Canada,

http://imps.mcmaster.ca/na-mkm-2002/.

[12] Flener, P., *Logic Program Schemata: Synthesis and Analysis*. Technical Report BU-CEIS-9502, Bilkent University, Ankara (Turkey), 1995.

[13] P.Flener, K.K. Lau, M.Ornaghi, *Correct-schema-guided Synthesis of Steadfast Programs*. In "Proceedings of the Twelfth IEEE International Automated Software Engineering Conference", pp. 153-160, IEEE Computer Society, 1997.

[14] Kraan, I., "Logic Program Synthesis via Proof Planning". Phd. thesis, Department of Artificial Intelligence, University of Edinburgh, 1993.

[15] Schmidt, D.C., R.E. Johnson, M. Fayad, *Software Patterns*. Guest editorial for the Communications of the ACM, Special Issue on Patterns and Pattern Languages, **Vol. 39**, No. 10, October 1996.

[16] Smith, D.R., *Top-Down Synthesis of Divide-and-Conquer Algorithms*. Artificial Intelligence **27(1)**(1985),pp. 43-96. (Reprinted in "Readings in Artificial Intelligence and Software Engineering", Eds. C. Rich and R. Waters, Morgan Kaufmann Pub. Co., Los Altos, CA, 1986, pp. 35-61).

[17] Smith, D.R., *KIDS: A semiautomatic program development system*. IEEE Transactions on Software Engineering Special Issue on Formal Methods

in Software Engineering **16(9)**(M. Kerber and M. Kohlhase (eds.))(1990), pp.1024-1043.

[18] Smith, D.R., *Mechanizing the Development of Software.* In ”Calculational System Design, Proceedings of the NATO Advance Study Institute”, Eds. M. Broy and R. Steinbrueggen, IOS Press, Amsterdam, 1999, pp. 251-292.

[19] Srinivas, Y.V., R. Jllig. *Specware: Formal support for composing software*. In ”Proceedings of the Conference on Mathematics of Program Construction”, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399-422.

[20] http://hillside.net/patterns

Appendix

Definitions

$$\underset{is\text{–}tuple[X]}{\forall}\left(is\text{–}sorted\text{–}version[X,Y] \Leftrightarrow \begin{cases} is\text{–}tuple[Y] \\ X \approx Y \\ is\text{–}sorted[Y] \end{cases}\right),$$

$$is\text{–}sorted[\langle\rangle],$$

$$\underset{x}{\forall}\, is\text{–}sorted[\langle x\rangle],$$

$$\underset{x,y,\bar{z}}{\forall}\left(is\text{–}sorted[\langle x,y,\bar{z}\rangle] \Leftrightarrow \begin{cases} x \geq y \\ is\text{–}sorted[\langle y,\bar{z}\rangle] \end{cases}\right),$$

$$\langle\rangle \approx \langle\rangle,$$

$$\underset{y,\bar{y}}{\forall}\langle\rangle \not\approx \langle y,\bar{y}\rangle,$$

$$\underset{x,\bar{x},\bar{y}}{\forall}(\langle x,\bar{x} \approx \langle\bar{y}\rangle\rangle \Leftrightarrow (x \in \langle\bar{y}\rangle \wedge \langle\bar{x}\rangle) \approx dfo[x,\langle\bar{y}\rangle]),$$

$$\underset{x}{\forall} x \notin \langle\rangle,$$

$$\underset{x,y,\bar{y}}{\forall}(x \in \langle y,\bar{y}\rangle) \Leftrightarrow ((x = y) \vee (x \in \langle y,\bar{y}\rangle)),$$

$$\underset{a}{\forall} dfo[a,\langle\rangle] = \langle\rangle,$$

$$\underset{a,x,\bar{x}}{\forall} dfo[a,\langle x,\bar{x}\rangle] = \begin{cases} \langle\bar{x}\rangle & \Leftarrow x = a \\ x \smile dfo[a,\langle\bar{x}\rangle] & \text{otherwise} \end{cases},$$

$$\underset{\bar{y}}{\forall}\langle\rangle \not\succ \langle\bar{y}\rangle,$$

$$\underset{x,\bar{x}}{\forall}\langle x,\bar{x}\rangle \succ \langle\rangle,$$

$$\underset{x,\bar{x},y,\bar{y}}{\forall} \langle x,\bar{x}\rangle \succ \langle y,\bar{y}\rangle \Leftrightarrow \langle \bar{x}\rangle \succ \langle \bar{y}\rangle,$$

$$\underset{x,\bar{y}}{\forall}(x \smile \langle \bar{y}\rangle = \langle x,\bar{y}\rangle),$$

$$\underset{X}{\forall}(\textit{is–tuple}[X] \Leftrightarrow \underset{\bar{x}}{\exists}(X = \langle \bar{x}\rangle)),$$

$$\underset{X}{\forall}(\textit{is-empty-tuple}[X] \Leftrightarrow (X = \langle\rangle)),$$

$$\underset{X}{\forall}(\textit{is-singleton-tuple}[X] \Leftrightarrow \underset{x}{\exists}(X = \langle x\rangle)),$$

$$\underset{X}{\forall}(\textit{is–trivial–tuple}[X] \Leftrightarrow (\textit{is-empty-tuple}[X] \vee \textit{is-singleton-tuple}[X]).$$

Axioms

$$\underset{x,\bar{x},y,\bar{y}}{\forall}(\langle x,\bar{x}\rangle = \langle y,\bar{y}\rangle) \Leftrightarrow ((x = y) \wedge (\langle \bar{x}\rangle = \langle \bar{y}\rangle)),$$

$$\underset{x,\bar{x}}{\forall} \langle x,\bar{x}\rangle \neq \langle\rangle.$$

Properties

$$\underset{\textit{is–trivial–tuple}[X]}{\forall} \textit{is–sorted}[X],$$

$$\underset{\textit{is–trivial–tuple}[X],\textit{is–tuple}[Y]}{\forall}(X \approx Y \Leftrightarrow (X = Y)),$$

$$\underset{\textit{is–tuple}[X]}{\forall} X \approx X,$$

$$\underset{\textit{is–tuple}[X,Y]}{\forall}(X \approx Y \Rightarrow Y \approx X),$$

$$\underset{\textit{is–tuple}[X,Y,Z]}{\forall}((X \approx Y \wedge Y \approx Z) \Rightarrow X \approx Z),$$

$$\underset{\textit{is–tuple}[X,Y]}{\forall}(X \succ Y \Rightarrow Y \not\succ X),$$

$$\underset{\textit{is–tuple}[X,Y,Z]}{\forall}((X \succ Y \wedge Y \succ Z) \Rightarrow X \succ Z).$$