Algorithm Invention and Verification by Lazy Thinking

Bruno Buchberger
Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria
bruno.buchberger@jku.at

Acknowledgment: 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. My PhD student Adrian Craciun implemented the case study in the frame of the *Theorema* system starting from my earlier versions of the induction prover, the conjecture generation algorithm, and the cascade.

■ Abstract

In this paper, we study algorithm invention and verification as a specific variant of systematic theory exploration and propose the "lazy thinking paradigm" for inventing and verifying algorithms automatically; i.e., for a given predicate logic specification of the problem in terms of a set of operations (functions and predicates), the method produces an algorithm that solves the problem together with a correctness proof for the algorithm. In the ideal case, the only information that has to be provided by the user consists of the formal problem specification and a complete knowledge base for the operations that occur in the problem specification. The "lazy thinking paradigm" is characterized

- ○ by using a library of *algorithm schemes*
- ○ and by using the information contained in *failing attempts to prove the correctness* theorem for an algorithm scheme in order to invent sufficient requirements on the auxiliary functions in the algorithm scheme.

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

■ Introduction

Algorithm invention (program synthesis) has a long tradition, see [Basin 2003] for a recent survey. In this paper, we consider the systematic (computer-aided, automated) invention of algorithms as a specific part of the general problem of systematic (computer-aided, automated) theory exploration. Systematic theory exploration was introduced in [Buchberger 1999] as an alternative to the isolated theorem proving paradigm that prevailed in formal, computer-supportedmathematics during the past decades. In [Buchberger 2000] we proposed various approaches to systematic, computer-supported mathematical theory exploration. In particular, we introduced the "lazy thinking" paradigm for proving mathematical theorems. The main idea of this paradigm consists in using the information of failing proof attempts for conjecturing intermediate lemmata that will allow to continue with the proof. The proof of the lemmata is then, again, attempted and may lead to the invention of sub-lemmatauntil the "cascade" of this invention terminates successfully.

In this paper, we modify the lazy thinking paradigm for inventing correct algorithms instead of inventing theorems: For a given predicate logic specification of the problem in terms of a set of operations (functions and predicates), the method invents an algorithm that solves the problem and also, simultaneously, provides a correctness proof for the algorithm.

Roughly, the method proceeds as follows:

- The method tries out, one after the other, various "algorithm schemes" (or "algorithm types") that are stored in a library of algorithm schemes for the given mathematical domain (or "data type"). 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 having this scheme.

- For the chosen algorithm type, the proof method is called for proving the correctness theorem. Typically, this proof will fail because nothing is known about the unspecified subalgorithms.

- >From the failing proof situation, by a conjecture generating algorithm, lemmata are generated that would enable the prover to complete the proof successfully. The lemmata will describe certain requirements on the subalgorithms. 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 described in the lemmata 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 subalgorithms 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−supportedsystems* 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 theory exploration and, in particular, algorithm invention and verification came to me while I was preparing a course on mathematical proving for high−schoolteachers in October 2001.)

In the sequel, we will illustrate the method by a case study, namely the automated synthesis of the merge−sort algorithm. The case study will be executed in the frame of the *Theorema* system. In particular, all occurring predicate logic formulae will be given in the *Theorema* syntax, see [Buchberger et al. 1997]. The case study will allow us also to explain some of the subtle details of the method.

Acknowledgement: My PhD student Adrian Craciun implemented the case study in the frame of the *Theorema* system starting from my earlier versions of the induction prover, the conjecture generation algorithm, and the cascade.

■ The Theorem Automatically Invented by the Method

The power of the method is best understood by considering the theorem that is automatically *invented* (and not only proved) by the method:

▫ Relative Correctness Theorem for Merge-Sort

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

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', 'is-permuted-version', etc. (see Appendix) and 'Is-Correct-Merge-Sort-Algorithm' is defined as follows:

$$\underset{\text{special,merged,left-split,right-split,sorted}}{\forall} \text{Is-Merge-Sort-Algorithm[special, merged, left-split, right-split, sorted]}$$

$$\Leftrightarrow$$

$$\left\{\begin{array}{l}
\underset{\text{is-tuple[X]}}{\forall}\left(\text{sorted[X]} = \left\{\begin{array}{lll}\text{special[X]} & \Leftarrow & \text{is-trivial-tuple[X]} \\ \text{merged[sorted[left-split[X]], sorted[right-split[X]]]} & \Leftarrow & \text{otherwise}\end{array}\right.\right) \\[2ex]
\underset{\substack{\text{is-tuple[X]} \\ \text{is-trivial-tuple[X]}}}{\forall} (\text{special[X]} = X) \\[2ex]
\underset{\substack{\text{is-tuple[X]} \\ \neg\text{is-trivial-tuple[X]}}}{\forall} \left\{\begin{array}{l}\text{left-split[X]} \prec X \\ \text{is-tuple[left-split[X]]} \\ \text{right-split[X]} \prec X \\ \text{is-tuple[right-split[X]]}\end{array}\right. \\[2ex]
\underset{\text{is-tuple[Y,Z]}}{\forall} \text{is-tuple[merged[Y, Z]]} \\[1ex]
\underset{\substack{\text{is-tuple[X,Y,Z]} \\ \neg\text{is-trivial-tuple[X]}}}{\forall} \left(\left\{\begin{array}{l}\text{left-split[X]} \approx Y \\ \text{right-split[X]} \approx Z \\ \text{is-sorted[Y]} \\ \text{is-sorted[Z]}\end{array}\right. \Rightarrow \left\{\begin{array}{l}\text{merged[Y, Z]} \approx X \\ \text{is-sorted[merged[Y, Z]]}\end{array}\right.\right)
\end{array}\right.$$

▫ Explanation

The theorem says that,

if

- the predicate ’is-sorted-version’ and its sub−operationssatisfy 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−trivialarguments, 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!

■ The Knowledge on the Problem

When attempting to solve a problem by an algorithm, we assume, of course, that the problem is "completely understood". In fact, it is a heuristic rule that "the better you understand the problem the closer you are to finding a solution". Generally speaking, the problem of sorting is an instance of a "problem scheme" (or "problem type") which we call "explicit problems". An explicit problem is given by a (binary) predicate P (called "problem specification") and the solution of the problem consists in finding a function f (called the "solution function" or "solution algorithm" in case f is an algorithmic function) such that

$$\underset{\text{is-object}[X]}{\forall} P[X, f[X]].$$

Here, ’is-object’ is a unary predicate that characterizes the objects in the data domain considered.

(Of course, explicit problems can also be defined for more than one input argument.)

In our case study, the problem specification is given by the binary predicate ’is-sorted-version’ which is defined as follows

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

Note that the input X is restricted to tuples, i.e. in our case, the input domain is the domain of tuples. Also note that the requirement that the output Y is a tuple is part of the problem specification. This is appropriate because, since the predicate is used for formulating the correctness theorem

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

the domain requirement on the input need not (and should not) be mentioned in the problem specification whereas the domain requirement on the output is an essential part of the problem specification.

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

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

and

$$\langle\rangle \approx \langle\rangle$$
$$\underset{y,\bar{y}}{\forall} \langle\rangle \not\approx \langle y, \bar{y}\rangle$$
$$\underset{x,\bar{x},\bar{y}}{\forall} (\langle x, \bar{x}\rangle \approx \langle\bar{y}\rangle \Leftrightarrow (x \in \langle\bar{y}\rangle \wedge \langle\bar{x}\rangle \approx \text{dfo}[x, \langle\bar{y}\rangle])).$$

(For the "sequence variables" notation '$\bar{x}$' etc., see the papers on *Theorema*, e.g. [Buchberger et al. 1997]. Sequence variables can be replaced by arbitrarily many terms. We use angle brackets as constructors for tuples: for example, ⟨2,2,3,1,4⟩ 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 occurence') that must be defined in terms of other auxilary 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 'Knowledge[sorted-version-of]', see appendix.

(Later in the paper, we will discuss the question of how to determine which properties of the operations between the problem specifying predicate 'is-sorted-version' and the basic operations on tuples should be included into the knowledge base.)

■ Algorithm Schemes

An "algorithm scheme" (or "algorithm type") for a given data type, in our view,

- ○ is a recursive definition of an unspecified "main" operation in terms of other unspecified "auxiliary" operations and the basic operations of the data type

- ○ together with a proof method that corresponds to the recursive definition in a natural way.

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

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

where you should think about the "main functions" '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. Hence, for didactic considerations, it might be better to just give them names like ’s’, ’sp’ ’m’, ’l’, and ’r’. However, the "semantic" function names above have the didactic advantage of suggesting where we are ultimately driving at.

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 "type requirement" on the auxiliary functions as a part of the algorithm scheme:

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

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

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

$$\underset{\substack{\text{is-tuple}[X] \\ \neg\text{is-trivial-tuple}[X]}}{\forall} \bigwedge \begin{cases} \text{left-split}[X] \prec X \\ \text{right-split}[X] \prec X \end{cases}$$

as a part of the recursive definition, which guarantees termination of the algorithm. (For ’$\prec$’ read ’has shorter length’, 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,
>
> $$\underset{\text{is-tuple}[X]}{\forall} A[X]$$
>
> it suffices to prove, for an arbitrary but fixed $\overline{x0}$,
>
> $$A[\langle\overline{x0}\rangle]$$
>
> under the assumptions
>
> $$\text{is-tuple}[\langle\overline{x0}\rangle]$$
>
> and
>
> $$\underset{\substack{\text{is-tuple}[Y] \\ Y \prec \langle\overline{x0}\rangle}}{\forall} A[Y].$$

This particular induction method (w.r.t. to the particular predicate $\prec$, ’shorter in length’, defined in the knowledge base) is based on the property that $\prec$ is a Noetherian relation. We do not include this property into the knowledge base. Rather, this property is implicitly used by allowing this induction method.

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 scheme libraries" that can then be used, in the way which we demonstrate in this paper, for inventing concrete algorithms for concrete problems. (In an analogous way, future mathematical knowledge management systems, should provide "problem scheme libraries", "data scheme libraries", "knowledge scheme libraries", and "definition scheme libraries". We cannot go into more details about this more general view in this paper but will expand on this aspect in forthcoming papers.)

We believe that, actually, for a given data type there exist only a few interesting algorithm types (algorithm schemes). These algorithm schemes should be put into libraries and can serve as an important input to algorithm invention systems. Another example of an algorithm scheme for algorithms on tuples is

$$s[\langle\rangle] = c$$
$$\underset{x,\overline{x}}{\forall} (s[\langle x, \overline{x}\rangle] = m[x, s[\langle\overline{x}\rangle]])$$

with the type requirement

$$\underset{x,\text{is-tuple}[Y]}{\forall} \text{is-tuple}[m[x, Y]]$$

and the following special induction method:

In order to prove, for an arbitrary property A,

$$\underset{\text{is-tuple}[X]}{\forall} A[X]$$

it suffices to prove

$$A[\langle\rangle]$$

and to prove, for arbitrary but fixed x0, $\overline{x0}$,

$$A[\langle x0, \overline{xo}\rangle]$$

under the assumption

$$A[\langle\overline{xo}\rangle].$$

■ Inventing the Algorithm 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', see appendix).

- We have chosen an algorithm scheme from a finite library of algorithm schemes for the domain of tuples (in our case: the "divide–and–conquer"algorithm scheme; note that we could start from any other scheme!). Remember that the scheme consists of a scheme for a induction function definition (including also type requirements for the auxilary functions) and a corresponding inductive proof method.

We now do 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

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

- Of course, this proof cannot succeed because basically nothing interesting 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 functions that would make it possible to get over the failing proof situation.
- 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.

Example:

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–foundedinduction w.r.t. $\succ$ on X:

We assume

$$\text{is-tuple}[\langle\overline{x o}\rangle]$$

and the induction hypothesis

$$\underset{\substack{\text{is-tuple}[Y]\\ \langle\overline{x o}\rangle \succ Y}}{\forall} \text{is-sorted-version}[Y, \text{sorted}[Y]]$$

and we show

$$\text{is-sorted-version}[\langle\overline{x o}\rangle, \text{sorted}[\langle\overline{x o}\rangle]].$$

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

CASE

$$\text{is-trivial-tuple}[\langle\overline{x o}\rangle]:$$

In this case, we have to show

$$\text{is-sorted-version}[\langle\overline{x o}\rangle, \text{special}[\langle\overline{x o}\rangle]]$$

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

(G1) $\text{is-tuple}[\text{special}[\langle\overline{x o}\rangle]],$

(G2) $\text{special}[\langle\overline{x o}\rangle] \approx \langle\overline{x o}\rangle,$

(G2) $\text{is-sorted}[\text{special}[\langle\overline{x o}\rangle]].$

(G1) is true because of the type requirement for 'special'.

For (G2), by the fact that

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

it suffices to prove that

$$\text{special}[\langle \overline{\text{xo}} \rangle] = \langle \overline{\text{xo}} \rangle.$$

Here we are stuck.

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, the *Theorema* proof refers to formulae in the knowledge base by labels, more specifically by hyperlinks, and we prefer not to use labels in the presentation of proofs in this paper for increasing readability.)

Now we analyze the failing proof situation and find:

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

$$\text{is-trivial-tuple}[\langle \overline{\text{xo}} \rangle].$$

- We have the temporary goal:

$$\text{special}[\langle \overline{\text{xo}} \rangle] = \langle \overline{\text{xo}} \rangle.$$

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

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

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.

■ Inventing the Algorithm by Lazy Thinking: Second Round

We now do exactly the same proof attempt once more. (Alternatively, we could jump back into the proof to the situation in which the first attempt failed. Both strategies, going back to the beginning and jumping right to the failing situation, have its advantages and disadvantages: Going back to the beginning may, in some examples, ultimately yield shorter proofs and jumping right to the failing situation, of course, saves proving effort.)

Since we have added a requirement on the auxiliary function 'special' we will be able to 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.

Example:

In the example, the next 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–foundedinduction w.r.t. $>$ on X:

We assume

$$\text{is-tuple}[\langle \overline{xo} \rangle]$$

.... *exactly as in the first proof attempt ...*

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

CASE

$$\text{is-trivial-tuple}[\langle \overline{xo} \rangle] :$$

In this case, by we have to show

$$\text{is-sorted-version}[\langle \overline{xo} \rangle, \text{special}[\langle \overline{xo} \rangle]]$$

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

$$\text{(G1)} \quad \text{is-tuple}[\text{special}[\langle \overline{xo} \rangle]],$$

$$\text{(G2)} \quad \text{special}[\langle \overline{xo} \rangle] \approx \langle \overline{xo} \rangle,$$

$$\text{(G2)} \quad \text{is-sorted}[\text{special}[\langle \overline{xo} \rangle]].$$

(G1) is true because of the type requirement for 'special'. (G2) is true because of

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

and the new requirement

$$\underset{\text{is-trivia-tuple}[X]}{\forall} (\text{special}[X] = X).$$

(G3) is true because of the same requirement and the following property of 'sorted'

$$\underset{\text{is-trivia-tuple}[X]}{\forall} \text{is-sorted}[X].$$

CASE

$$\neg\ \text{is-trivial-tuple}[\langle \overline{xo} \rangle] :$$

In this case, we have to show

$$\text{is-sorted-version}[\langle \overline{xo} \rangle, \text{merged}[\text{sorted}[\text{left-split}[\langle \overline{xo} \rangle], \text{sorted}[\text{right-split}[\langle \overline{xo} \rangle]]]]].$$

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

$$\text{(H1)} \quad \text{is-tuple}[\text{merged}[\text{sorted}[\text{left-split}[\langle \overline{xo} \rangle], \text{sorted}[\text{right-split}[\langle \overline{xo} \rangle]]]]],$$

$$\text{(H2)} \quad \langle \overline{xo} \rangle \approx \text{merged}[\text{sorted}[\text{left-split}[\langle \overline{xo} \rangle], \text{sorted}[\text{right-split}[\langle \overline{xo} \rangle]]]],$$

$$\text{(H3)} \quad \text{is-sorted}[\text{merged}[\text{sorted}[\text{left-split}[\langle \overline{xo} \rangle], \text{sorted}[\text{right-split}[\langle \overline{xo} \rangle]]]]].$$

>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

$$\text{is-sorted-version}[\text{left-split}[\langle \overline{xo} \rangle], \text{sorted}[\text{left-split}[\langle \overline{xo} \rangle]]],$$

$$\text{is-sorted-version}[\text{right-split}[\langle \overline{xo} \rangle], \text{sorted}[\text{right-split}[\langle \overline{xo} \rangle]]].$$

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

(AL1) $\text{is-tuple}[\text{sorted}[\text{left-split}[\langle\overline{x_o}\rangle]]]$,

(AL2) $\text{left-split}[\langle\overline{x_o}\rangle] \approx \text{sorted}[\text{left-split}[\langle\overline{x_o}\rangle]]$,

(AL3) $\text{is-sorted}[\text{sorted}[\text{left-split}[\langle\overline{x_o}\rangle]]]$,

(AR1) $\text{is-tuple}[\text{sorted}[\text{right-split}[\langle\overline{x_o}\rangle]]]$,

(AR2) $\text{right-split}[\langle\overline{x_o}\rangle] \approx \text{sorted}[\text{right-split}[\langle\overline{x_o}\rangle]]$,

(AR3) $\text{is-sorted}[\text{sorted}[\text{right-split}[\langle\overline{x_o}\rangle]]]$.

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

Now we are stuck.

Proof Attempt End

Now we analyze the failing proof situation and find:

- We have the case assumption and the formulae (AL1), ..., (AR3) as termporary 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 do this automatically) that the following requirement on the functions 'left–split', 'right–split'und 'merged'

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

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.

■ Inventing the Algorithm by Lazy Thinking: Last Round

We now do exactly the same proof attempt once more (or we just jump to the proof situation where the previous proof attempt got stuck.)

This time, the inductive proof will succeed using the added requirement on 'left–split','right–split'und 'merged' for proving (H2) and (H3).

If we now collect the requirements on the functions'special','left–split','right–split',and'merged',wesee that we invented and proved the "Relative Correctness Theorem for Merge–Sort"formulated at the beginning of this paper.

■ Automation of the Lazy Thinking Procedure

The "lazy thinking" procedure for inventing algorithms together with their correctness proofs, first of all, is meant to be a heuristic guide for human invention and verification.

However, the procedure can be made completely automatic (algorithmic) if we manage

- to automate proving in the specific area and
- to automate generating conjectures (requirements on auxiliary functions) from the temporary assumptions and the left−overgoals 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 numerous problems on tuples. Inductive provers that qualify for the use in the frame of the lazy thinking algorithm invention procedure must have a couple of properties: First, they must prove theorems in a "natural style" that proceeds from proof situations with temporary assumptions and goals to other such proof situations. Second, they must generate a proof object also in case of failing proofs. This is so because the essence of the lazy thinking method is "learning from failures".

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⇒ 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⇒G) by variables v,... and produces the conjecture $\underset{v,\ldots}{\forall}(A\Rightarrow G)$. By this strategy, one can produce, for example, the conjecture (requirement)

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

  in the first round above.

- The second strategy can handle failing proof situations in the induction step parts of proofs. It first, again, replaces all "arbitrary but fixed" constants in (A⇒G) by variables v,.... Then it looks for terms whose head is the function constant for the algorithm to be synthesized. (In our case, this is the function constant 'sorted'.) All these terms are then replaced by new variables w,... and then the variant $\underset{v,\ldots,w,\ldots}{\forall}(A\Rightarrow G)$ is taken as the new conjecture. By this strategy, one can produce, for example, the conjecture (requirement)

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

  in the second round above.

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

```
Prove[Theorem["correctness of sorting"],
  using → Theory["sorting"],
  by → Cascade[SqnsEqCasePC, GenerateConjectures]
```

Here, Theory["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",

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

$$\text{is-sorted}[\langle\rangle]$$

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

... (*all formulae in the appendix*) ...]

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

Theorem["correctness of sorting",

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

'SqnsEquCasePC' 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

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

$$\underset{\substack{\text{is-tuple}[X]\\ \neg\text{is-trivial-tuple}[X]}}{\forall}\ \text{is-tuple}[\text{left-split}[X]]$$

.... *etc* ....,

to the knowledge and organizes the main loop of the proof by the particular induction scheme.

We are now working on a generale 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, the prover can attempt various algorithm syntheses successively without user interaction in between.

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 with 1.8 GHz, the user will get

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

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

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

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

Now the user knows that the divide–and–conquealgorithm is a correct sorting program if one uses auxiliary functions 'special', 'left-split', 'right-split', and 'merged' that satisfy the type requirements and the above, synthesized, requirements. In other words, the user does not only get one particular sorting algorithm synthesized (together with a correctness proof) but gets a whole spectrum of possible correct sorting algorithms!

We can now proceed in two ways:

- Either we already have functions 'left−split','right−split',and 'merged' (possibly with other names) in our knowledge base that satisfy the requirements. (The proof that the functions in the knowledge base satisfy the requirement should be something current automated provers can do. See, however, next section.) Then we can use them as auxiliary functions and we are done, i.e. we have a correct sorting algorithm, which now can be executed. In fact, in *Theorema*, the execution of algorithms, i.e. the application of algorithms to concrete inputs, can be done within the *Theorema* system itself, i.e. proving and computing can be done in the same language and logic! (In other words, part of the inference mechanism of the logic is used as the interpreter of a universal programming language.) We will show this by an example below.

- Or we take the type requirements and the synthesized requirements on 'special', 'left-split', 'right-split', and 'merged' as new specifications for synthesizing appropriate functions again by the lazy thinking procedure. The requirement for 'special' is easy to fulfil: In fact, the requirement itself is a suitable function definition for 'special'. The requirements for 'left-split', 'right-split', and 'merged' are intertwined. They do not constitute an "explicit" problem specification. In principle, it is possible to apply the lazy thinking procedure also for synthesizing algorithms whose specification is not in explicit form. However, if possible, it is much better to try to decouple interwined specifications before one starts to synthesize algorithms that meet the specification.

In our case, it is in fact possible to replace the intertwined specification for 'left-split', 'right-split', and 'merged' by a decoupled one. Namely, it can be (automatically) shown that the following decoupled specification entails the above intertwined specification:

$$\underset{\substack{\text{is-tuple}[X]\\ \neg\text{is-trivial-tuple}[X]}}{\forall} (\text{left-split}[X] \asymp \text{right-split}[X]) \approx X$$

$$\underset{\text{is-tuple}[Y,Z]}{\forall}\left(\left\{\begin{array}{l}\text{is-sorted}[Y]\\ \text{is-sorted}[Z]\end{array}\right. \Rightarrow \left\{\begin{array}{l}\text{merged}[Y, Z]\approx(Y\asymp Z)\\ \text{is-sorted}[\text{merged}[Y, Z]]\end{array}\right.\right)$$

(Here, '≍' denotes concatentation.) Using the lazy thinking procedure on this specification working with the algorithm scheme

$$\text{merged}[\langle\rangle, \langle\rangle] = \text{mee}$$
$$\underset{y,\overline{y}}{\forall}\ \text{merged}[\langle\rangle, \langle y, \overline{y}\rangle] = \text{meg}[y, \overline{y}]$$
$$\underset{x,\overline{x}}{\forall}\ \text{merged}[\langle x, \overline{x}\rangle, \langle\rangle] = \text{mge}[x, \overline{x}]$$
$$\underset{x,\overline{x},y,\overline{y}}{\forall}\ \text{merged}[\langle x, \overline{x}\rangle, \langle y, \overline{y}\rangle] = \left\{\begin{array}{l}\text{mgg1}[x, \text{merged}[\langle\overline{x}\rangle, \langle y, \overline{y}\rangle]] \Leftarrow p[x, y]\\ \text{mgg2}[y, \text{merged}[\langle x, \overline{x}\rangle, \langle\overline{y}\rangle]] \Leftarrow \neg\, p[x, y]\end{array}\right\}$$

where 'mee', 'meg', 'mge', 'mgg1', 'mgg2' and 'p' are the unknown auxiliary operations, yields the usual merge algorithm

$$\text{merged}[\langle\rangle, \langle\rangle] = \langle\rangle$$
$$\underset{y,\overline{y}}{\forall}\ \text{merged}[\langle\rangle, \langle y, \overline{y}\rangle] = \langle y, \overline{y}\rangle$$
$$\underset{x,\overline{x}}{\forall}\ \text{merged}[\langle x, \overline{x}\rangle, \langle\rangle] = \langle x, \overline{x}\rangle$$
$$\underset{x,\overline{x},y,\overline{y}}{\forall}\ \text{merged}[\langle x, \overline{x}\rangle, \langle y, \overline{y}\rangle] = \left\{\begin{array}{l}x \smile \text{merged}[\langle\overline{x}\rangle, \langle y, \overline{y}\rangle] \Leftarrow x > y\\ y \smile \text{merged}[\langle x, \overline{x}\rangle, \langle\overline{y}\rangle] \Leftarrow \neg\, x > y\end{array}\right\}$$

Similarly, concrete algorithms that satisfy the specification of 'left-split' and 'right−split'can be synthesized, for example

$$\text{left-split}[\langle\rangle] = \langle\rangle$$
$$\underset{x}{\forall}\,(\text{left-split}[\langle x\rangle] = \langle x\rangle)$$
$$\underset{x,y,\overline{z}}{\forall}\,(\text{left-split}[\langle x, y, \overline{z}\rangle] = x \smile \text{left-split}[\langle\overline{z}\rangle])$$
$$\text{right-split}[\langle\rangle] = \langle\rangle$$
$$\underset{x}{\forall}\,(\text{right-split}[\langle x\rangle] = \langle\rangle)$$
$$\underset{x,y,\overline{z}}{\forall}\,(\text{right-split}[\langle x, y, \overline{z}\rangle] = y \smile \text{right-split}[\langle\overline{z}\rangle])$$

Note that these algorithms 'merged', 'left-split', and 'right-split' are now "concrete" in the sense that they only need auxiliary operations that are basic operations on tuples (and the ordering predicate '>' on the objects in tuples), i.e. no more synthesis step is necessary.

Putting all these definitions into one theory by

$$\textbf{Theory}\Big[\text{"sorting"},\; \underset{\text{is-tuple}[X]}{\forall}\left(\text{sorted}[X] = \begin{cases}\text{special}[X] & \Leftarrow \text{is-trivial-tuple}[X]\\ \text{merged}[\text{sorted}[\text{left-split}[X]], \text{sorted}[\text{right-split}[X]]] & \Leftarrow \text{otherwise}\end{cases}\right)$$
$$\text{merged}[\langle\rangle, \langle\rangle] = \langle\rangle$$
$$\dots\ \dots$$
$$\underset{x,y,\overline{z}}{\forall}\,(\text{right-split}[\langle x, y, \overline{z}\rangle] = y \smile \text{right-split}[\langle\overline{z}\rangle])\Big]$$

One can now compute within *Theorema.* For example, entering

Compute[sorted[⟨1, 233, 3, 44, 5, 66, 7, 8⟩]]

yields

⟨233, 66, 44, 8, 7, 5, 3, 1⟩.

(Also the definitions of the basic operations on tuples must be made part of Theory['sorting'] or, alternatively, one can declare these operations as "built−in"in *Theorema,* see the papers on *Theorema.*)

■ 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. Seemlingly, 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, ..., and a requirement on f, ..., i.e. a formula R[f,...]
- ○ find operation names F, ... occurring in K such that R[F,...] is a logical consequences of K.

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

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

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

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

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

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

$$L[\langle\rangle] = \langle\rangle$$

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

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

$$R[\langle\rangle] = \langle\rangle$$

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

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

and the following requirement R[left-split, right-split, merged]

$$\underset{\substack{\text{is-tuple}[X] \\ \neg\text{is-trivial-tuple}[X]}}{\forall} \left\{\begin{array}{l} \text{left-split}[X] \prec X \\ \text{is-tuple}[\text{left-split}[X]] \\ \text{right-split}[X] \prec X \\ \text{is-tuple}[\text{right-split}[X]] \end{array}\right.$$

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

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

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 arbitrarily complicated.

"Complete" knowledge bases are knowledge bases in which, for all the occurring operations, all the possible "interactions" between the operations have already been studied resulting in "rewrite properties" of these operations. For example, for the operations L, R, M defined above the following interactions with the operations '$\approx$', $\asymp$', and 'is-sorted'

$$\underset{\substack{\text{is-tuple}[X] \\ \neg\text{is-trivial-tuple}[X]}}{\forall} \left\{\begin{array}{l} L[X] \prec X \\ \text{is-tuple}[L[X]] \\ R[X] \prec X \\ \text{is-tuple}[R[X]] \end{array}\right.$$

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

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

$$\underset{\text{is-tuple}[Y,Z]}{\forall} \left(\left\{\begin{array}{l} \text{is-sorted}[Y] \\ \text{is-sorted}[Z] \end{array}\right. \Rightarrow \left\{\begin{array}{l} M[Y, Z] \approx (Y \asymp Z) \\ \text{is-sorted}[M[Y, Z]] \end{array}\right.\right)$$

and many other such interactions should already be available in the knowledge base (i.e. it should have been proved in a "complete exploration" phase of the knowledge base). Then the proof that als R[L, R, M] holds is "relatively easy", namely it can be done essentially by rewriting and other simple proof techniques ("symbolic computation proof techniques", "high–schoolproving", i.e. proving without quantifiers).

In other words, one could define that a knowledge base is "complete" iff proving properties that are not yet in the knowledge base is possible by "basic proving" (i.e. proving esssentially without quantifiers). Mathematical knowledge bases should be complete in this sense so that "retrieving knowledge" can be done by basic proving. Of course, all this is vague terminology. However, we think that this points into the right direction and we will elaborate on this philosophy in some other paper.

■ A Functorial View of Program Synthesis

We have seen that, by the above "lazy thinking" approach, algorithms A involving auxiliary operations B, C, ... can be synthesized that meet their specification P under the assumption that the ingredient auxiliary operations B, C, ... meet a certain other specification Q. In other words, the synthesis procedure invents and proves a theorem of the following structure

$$\begin{array}{l}\text{knowledge}[P]\\ \Rightarrow\\ \underset{B,C,\ldots}{\forall}\left(\left\{\begin{array}{l}\underset{X}{\forall}(A[X]=F[X,B,C,\ \ldots])\\ Q[B,C,\ ..]\end{array}\right. \Rightarrow \underset{X}{\forall} P[X,A[X]]\right)\end{array}$$

where F is the scheme (the "functional") which we use in order to define A in terms of the auxiliary operations B, C, ...

For all this,we assumed that the specification P of the algorithm to be synthesized is "completely" given, whatever this means.

We now can go one step further: After the synthesis is completed (resulting both in the specification Q and in the proof of $\underset{X}{\forall} P[X, A[X]]$, we can analyze which properties of P and its auxiliary operations actually entered into the correctness proof. Doing this, often results in a much more general theorem: If K[P,p,q,...] is the knowledge on P and its ingredient operations p, q, ... that is really needed in the correctness proof, then we may state that

$$\begin{array}{l}\underset{P,p,q,\ldots}{\forall}\left(K[P,p,q,\ \ldots]\right.\\ \quad\Rightarrow\\ \quad\left.\underset{B,C,\ldots}{\forall}\left(\left\{\begin{array}{l}\underset{X}{\forall}(A[X]=F[X,B,C])\\ Q[B,C,\ ..]\end{array}\right. \Rightarrow \underset{X}{\forall} P[X,A[X]]\right)\right)\end{array}$$

Carrying out this analysis in the above example, yields the following theorem:

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

where 'Is-Merge-Sort-Algorithm' is defined as in the section on the correctness of merge–sortand 'Is-Sorting-Problem' is defined as follows:

$$\underset{\text{is-sorted-version},\approx,\text{is-sorted}}{\forall}\ \text{Is-Sorting-Problem}[\text{is-sorted-version},\ \approx,\ \text{is-sorted}]$$

$$\Leftrightarrow$$

$$\begin{cases} \underset{\text{is-tuple}[X]}{\forall} \left(\text{is-sorted-version}[X, Y] \Leftrightarrow \begin{cases} \text{is-tuple}[Y] \\ X \approx Y \\ \text{is-sorted}[Y] \end{cases} \right) \\ \underset{\text{is-trivia-tuple}[X]}{\forall} \text{is-sorted}[X] \\ \underset{\text{is-trivial-tuple}[X],\text{is-tuple}[Y]}{\forall} (X \approx Y \Leftrightarrow (X = Y)) \\ \text{is-noetherian}[\succ] \end{cases}$$

Note that the main omission in this knowledge base is the concrete definition of 'is-sorted' and '≈' in terms of more elementary operations. Hence, we can read the theorem which we invented by the lazy thininking procedure also in the following way:

Consider the "functor"

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

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

that expands a domain containing the operations '≻', '≈', 'is-sorted', 'special', 'merged', 'left-split', and 'right-split' by the new operations 'is-sorted' and 'sorted'. Then we can be sure that the function 'sorted' is a correct algorithm for the explicit problem 'is-sorted-version' as long as the operations ' '≻' etc. satisfy the properties given in the definitions of 'Is-Sorting-Problem' and 'Is-Merge-Sort-Algorithm'.

The power of this theorem can best be appreciated if you replace the semantic names 'is-sorted' etc. by some arbitrary constants like 'p', etc.

Similar ideas, maybe less explicit, have been expressed in [Farmer 2003] and [Schwarzweller 2003].

One also may view algorithm synthesis as higher-ordersolving. For example, in the problem of sorting we want to find a function 'sorted' that satisfies the specification

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

in the theory of 'is-sorted-version', i.e. in the theory compiled in the appendix or a sub-theorythereof. Now, in our setting, we do not specify 'sorted' by a higher-orderterm but, rather, by the algorithm synthesis procedure we gradually expand the theory by suitable definitions of 'sorted' and its auxiliary operations. Of course, formally, the result could be re-writtenby a higher-orderterm. However, we think that our setting is more natural and the result is more readable.

■ 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
- can also be used for extracting minimal requirements for the concepts in the problem specification and
- can also be seen under the general perspective as viewing algorithm schemes and problem specifications as functors that transport requirements on the ingredient auxiliary operations into correctness theorems (stating that the algorithm defined by the scheme satisfies the problem specification)
- and can, thereby, also seen as a contribution to the problem of generating re−usablealgorithms.

Algorithms and theorems are only two sides of the same coin. Hence, algorithm and theorem invention and verification can be handled by the same approaches, e.g. the lazy thinking approach. In the case of theorem invention, knowledge schemes take over the role of algorithm schemes. Also, it should be clear that the invention of interesting notions and interesting problems for these notions is another important part of "mathematical knowledge management". The general role of algorithm schemes, knowledge schemes, problem schemes, definition schemes, and data schemes for mathematical knowledge management will be studied in another paper.

The general subject of "mathematical knowledge management" has first been taken up in the "1st International Workshop on Mathematical Knowledge Management", September 14−16,2001, organized by this author at RISC, see also the special issue [Buchberger et al. 2003], which contains some of the papers of this conference. Meanwhile mathematical knowledge management has seen increasing interest by the international research community. In our view, computer−supportedmathematical theory exploration will be the key technology for future mathematical knowledge management.

■ References

[Basin 2003] D. Basin, 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.

[Buchberger 1999] B. Buchberger. Theorem Proving Versus Theory Exploration. Invited talk at the Calculemus Workshop, Univ. of Trento, July 11, 1999, Italy.

[Buchberger 2000] B. Buchberger. 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.

[Buchberger et al. 1997] B. Buchberger, 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.

[Buchberger et al. 2003] B.Buchberger, G.Gonnet, M.Hazewinkel (eds.), Mathematical Knowledgement Management, special issue of the Journal Annals of Mathematics and Artificial Intelligence, Vol. 38, Nos. 1–3, Kluwer Academic Publishers, 2003.

[Farmer 2003] W. Farmer. A Formal Framework for Managing Mathematics. In [Buchberger et al. 2003].

[Schwarzweller 2003] C. Schwarzweller. Towards Formal Support for Generic Programming. Habilitation Thesis, University of Tübingen, Computer Science Department, 2003.

■ Appendix: Knowledge Base for the Sorting Problem

■ Definitions

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

$$\text{is-sorted}[\langle\rangle]$$

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

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

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

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

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

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

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

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

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

$$\underset{\bar{y}}{\forall}\ \neg\ \langle\rangle \succ \langle\bar{y}\rangle$$

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

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

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

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

$$\underset{X}{\forall}\left(\text{is-tuple}[X] \Leftrightarrow \underset{\overline{x}}{\exists}(X = \langle\overline{x}\rangle)\right)$$

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

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

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

■ Axioms

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

$$\underset{x,\overline{x}}{\forall}\langle x, \overline{x}\rangle \neq \langle\rangle$$

■ Properties

$$\underset{\text{is-trivial-tuple}[X]}{\forall}\text{is-sorted}[X]$$

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

$$\underset{\text{is-tuple}[X]}{\forall}X \approx X$$

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

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

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

$$\underset{\text{is-tuple}[X,Y,Z]}{\forall}((X \succ Y \wedge Y \succ Z) \Rightarrow X \succ Z)$$