Algorithm Retrieval: Concept Clarification and Case Study in *Theorema*

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

■ Abstract

Algorithm retrieval is a special case of mathematical knowledge retrieval, which is one of the fundamental problems of mathematical knowledge management.

In this paper, we distinguish between various versions of the problem of algorithm retrieval focusing on the version which can only be appropriately formulated in the frame of formal logic. This is the following problem:

> Given formulae S[p, q, ...] that specify properties of a couple of (algorithmic operations) p,q,... and a knowledge base K of formulae,
>
> find operations P, Q, ... (occurring in the vocabulary of K) such that S[P, Q, ...] is a logical consequence of K.

Considering candidate operations P, Q, ... in the vocabulary of K, the problem of checking whether or not S[P, Q, ...] is a logical consequence of K, can be trivial, easy, moderately difficult or very difficult depending on how much knowledge on P, Q, ... is already contained in K. Accordingly, in this paper, we consider the following two instances of algorithm retrieval in a formal (logic based) setting:

- K contains no knowledge on P, Q, ... except their (algorithmic) definitions. **In this case, algorithm retrieval essentially is algorithm verification**.
- K contains so much knowledge on P, Q, ... that the proof of S[P, Q, ...] from K becomes "easy" in the sense that it can be done by "**symbolic computation**" (conditional equational proving, propositional proving, manipulation with bounded quantifiers etc.). **This is the case which we consider to be desirable**, i.e. good knowledge bases should contain sufficiently much knowledge for making algorithm retrieval less cumbersome than algorithm verification. In other words, as a tendency, one should always try to "complete" mathematical knowledge bases on given concepts to the extent that subsequent proving (and disproving) of additional statements about these concepts is "easily" possible by symbolic computation proving.

Furthermore, it may be the case that K contains no candidate operations P, Q, ... for which S[P, Q, ...] is a logical consequence of K. **In this case, algorithm retrieval becomes algorithm invention (algorithm synthesis).**

We give a **case study** in which the distinction between the three basic cases and the dependence of algorithm retrieval on the contents of the knowledge base becomes clear. We demonstrate that **algorithm retrieval in a knowledge base essentially is theorem proving**, namely proving that the given specification of the algorithm(s), for algorithms that occur in the knowledge base, is a logical consequence of the information on these algorithms already stored in the knowledge base.

Keywords: algorithm retrieval, mathematical knowledge retrieval, symbolic computation proving, high-school proving, proving by rewriting, physicists' proving, basic prover, complete knowledge for mathematical notions, algorithm verification, algorithm synthesis, program synthesis, decoupling of requirements, re-usable algorithms, functors, requirement engineering, didactics of programming, sorting, merging, merge-sort, *Theorema*.

Acknowledgement: Sponsored by FWF (Österreichischer Fonds zur Förderung der Wissenschaftlichen Forschung; Austrian Science Foundation), project SFB 1302 ("*Theorema*") of the SFB 013 ("Scientific Computing").

■ Introduction

As any other information retrieval problem, the problem of algorithm retrieval is the problem of finding **detailed information** about an algorithm, e.g. code for the algorithm, under the assumption that **some information** about the algorithm, for example its name or its specification, is given.

Of course this problem is only meaningful if we also specify the **knowledge base** in which the search should be carried out: libraries with journals and books in printed form, mathematical software systems like Mathematica or Maple, or information in digitized form accessible through the web. Currently, quite some effort is spent on "digitizing" printed mathematical papers, to add semantic information to digitized versions of mathematical papers and to turn digitized papers into papers in a formal language, see [Buchberger et al. 2003] for various research directions in digitization of mathematical knowledge.

In this paper, we start from the situation that

- the given information on the algorithm,
- the knowledge base,
- and the information to be found

are **presented as formulae in a logical language**. We believe that this assumption will be realistic in the near future. In other words, we believe that only when mathematical information is available in completely formal presentation within a well-defined logic, the more sophisticated versions of the information retrieval problem can be attacked.

Of course, on a first layer of algorithm retrieval, one thinks about finding details of an algorithm by a search through libraries given some keywords about the algorithm like "Lagrange", "interpolation", "polynomials" etc. These days, the solution of this problem, with the web, powerful search engines, special research and citations indices etc., is already in a state that is by far more pleasant than only 10 years ago and more progress is to be expected soon by the joint effort of the mathematical knowledge management community that recently established itself also as a formal organization at the 1st International Workshop of Mathematical Knowledge Management (MKM) initiated by this author with annual successor conferences well established both in Europe and the US, see [Asperti 2003].

However, we need to go a decisive step forward: In this paper, the information on the algorithm(s) is not given by "external" keywords but is given by formulae that describe properties of the algorithm(s), which we call "specification" of the algorithm(s), and the problem consists in deciding whether an algorithm (algorithms) meeting the given specification are available in the given formulae knowledge base. Needless to say that this

problem is much more demanding than the algorithm retrieval problem based on textual information but one might argue whether this, more demanding, version of the algorithm retrieval problem really occurs "in pratice". Thus, let us start with an example that illustrates the substantial relevance of this formal version of algorithm retrieval:

Example: In our "lazy thinking" automated algorithm invention (program synthesis) paradigm based on algorithm schemes and learning by failure, see [Buchberger 2003a], we showed how finding an algorithm 'sorted' satisfying the specification

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

can be automatically reduced to finding algorithms 'left-split', 'right-split', and 'merged' satisfying the specification

$$\text{is-left} - \text{right} - \text{merge-structure[left-split, right-split, merged]}$$

defined by

$$\forall_{\text{left-split,right-split,merged}} \left| \begin{array}{l} \text{is-left-right-merge-structure[left-split, right-split, merged]} \Leftrightarrow \\ \\ \left(\begin{array}{l} \left\{ \begin{array}{l} \forall_{\substack{\text{is-tuple[X]} \\ \neg\text{is-trivial-tuple[X]}}} \left\{ \begin{array}{l} \text{left-split[X]} \prec \text{X} \\ \text{is-tuple[left-split[X]]} \\ \text{right-split[X]} \prec \text{X} \\ \text{is-tuple[right-split[X]]} \end{array} \right. \\ \forall_{\text{is-tuple[Y,Z]}} \text{is-tuple[merged[Y, Z]]} \end{array} \right\} \\ \\ \forall_{\substack{\text{is-tuple[X,Y,Z]} \\ \neg\text{is-trivial-tuple[X]}}} \left(\left\{ \begin{array}{l} \text{left-split[X]} \approx \text{Y} \\ \text{right-split[X]} \approx \text{Z} \\ \text{is-sorted[Y]} \\ \text{is-sorted[Z]} \end{array} \right. \Rightarrow \left\{ \begin{array}{l} \text{merged[Y, Z]} \approx \text{X} \\ \text{is-sorted[merged[Y, Z]]} \end{array} \right. \right) \end{array} \right) \end{array} \right.$$

Here, (all) details of the specification predicate 'is-sorted-version' must be contained in the underlying knowledge base, see appendix.

For completing the algorithm invention process we must, of course, find suitable algorithms 'left-split', 'right-split', and 'merged'. For this problem, it is not sufficient to do a textual search, in the knowledge base, for algorithms with names 'left-split', 'right-split', and 'merged':

- There may exist algorithms with these names in the knowledge base. However, they may not meet exactly the specification above (although their names may suggest this!).
- There may exist algorithms with these names in the knowledge base and these algorithms may well meet the specification above but their definitions and properties may, literally, be quite distinct from the specification above. Thus, by a purely textual search, we cannot find out that these algorithms are appropriate .

- There may exist algorithms whose names are quite distinct from the names above and these algorithms may well meet the specification above but, again, we cannot find this out by a purely textual search.

Hence, "finding" in the existing knowledge base suitable algorithms that meet the above specification is a problem that goes far beyond purely textual search of appropriate algorithm names. Rather, the problem is essentially a theorem proving problem, which we call "**logical algorithm retrieval problem**" and whose nature we will explore in the subsequent sections. We will study variants of the logical algorithm invention problem of increasing logical complexity.

Note: All examples, in this paper, are presented in the *Theorema* system, see [Buchberger et al. 1997, 2000]. However, the analysis of the algorithm retrieval problem presented is quite general and independent of a particular system for computer-supported theory exploration and also holds in the context of theory exploration without computer support.

■ The Trivial Variant: Logical Algorithm Retrieval = Formulae Identity

In the first variant of the logical algorithm retrieval problem, we assume that the specification for the algorithms to be found (in our example, the algorithms 'left-split', 'right-split', and 'merged') and inductive definitions for these algorithms already appear in the given knowledge base. In our example, this means that the formulae

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

$$\forall_{\text{is-tuple[Y,Z]}} (\text{is-tuple[merged[Y, Z]]})$$

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

and, for example, the formulae

$$\text{left-split}[\langle\rangle] = \langle\rangle$$

$$\forall_{\text{x}} (\text{left-split}[\langle \text{x} \rangle] = \langle \text{x} \rangle)$$

$$\forall_{\text{x,y,z}} (\text{left-split}[\langle \text{x, y, } \overline{\text{z}} \rangle] = \text{x} \frown \text{left-split}[\langle \overline{\text{z}} \rangle])$$

$$\text{right-split}[\langle\rangle] = \langle\rangle$$

$$\forall_{\text{x}} (\text{right-split}[\langle \text{x} \rangle] = \langle\rangle)$$

$$\forall_{\text{x,y,z}} (\text{right-split}[\langle \text{x, y, } \overline{\text{z}} \rangle] = \text{y} \frown \text{right-split}[\langle \overline{\text{z}} \rangle])$$

$$\text{merged}[\langle\rangle, \langle\rangle] = \langle\rangle$$

$$\forall_{\text{y, } \overline{\text{y}}} (\text{merged}[\langle\rangle, \langle \text{y, } \overline{\text{y}} \rangle] = \langle \text{y, } \overline{\text{y}} \rangle)$$

$$\forall_{\text{x, } \overline{\text{x}}} (\text{merged}[\langle \text{x, } \overline{\text{x}} \rangle, \langle\rangle] = \langle \text{x, } \overline{\text{x}} \rangle)$$

$$\forall_{\text{x, } \overline{\text{x}}, \text{y, } \overline{\text{y}}} \left(\text{merged}[\langle \text{x, } \overline{\text{x}} \rangle, \langle \text{y, } \overline{\text{y}} \rangle] = \begin{Bmatrix} \text{x} \frown \text{merged}[\langle \overline{\text{x}} \rangle, \langle \text{y, } \overline{\text{y}} \rangle] \Leftarrow \text{x} > \text{y} \\ \text{y} \frown \text{merged}[\langle \text{x, } \overline{\text{x}} \rangle, \langle \overline{\text{y}} \rangle] \Leftarrow \neg\, \text{x} > \text{y} \end{Bmatrix} \right)$$

that give inductive definitions of 'left-split', 'right-split', and 'merged', already appear in the knowledge base. (Here and in the rest of the paper we assume that the knowledge base only contains formulae whose correctness has already been (automatically) proved w.r.t. the underlying theory that introduces the data type in which we are working, in our case the theory of tuples.)

In this case, checking whether the algorithms 'left-split', 'right-split', and 'merged' meet the specification

$$\text{is-left-right-merge-structure}[\text{left-split}, \text{right-split}, \text{merged}]$$

is a trivial task in the exact sense that proving is a "one-line" proof.

Of course, still, this task would be trivial if the property of the algorithms 'left-split', 'right-split', and 'merged', in the knowledge base, were described by some logical variant (with different choice of the bound variables, with some re-arrangement of the formulae etc.), for example by the variant

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

$$\underset{\text{is-tuple}[B,C]}{\forall} \text{is-tuple}[\text{merged}[B, C]]$$

$$\underset{\substack{\text{is-tuple}[A,B,C]\\ \neg\text{is-trivial-tuple}[A]}}{\forall} \left(\begin{Bmatrix} \text{left-split}[A] \approx B \\ \text{right-split}[A] \approx C \\ \text{is-sorted}[B] \\ \text{is-sorted}[C] \end{Bmatrix} \Rightarrow \text{merged}[B, C] \approx A \right)$$

$$\underset{\substack{\text{is-tuple}[A,B,C]\\ \neg\text{is-trivial-tuple}[A]}}{\forall} \left(\begin{Bmatrix} \text{left-split}[A] \approx B \\ \text{right-split}[A] \approx C \\ \text{is-sorted}[B] \\ \text{is-sorted}[C] \end{Bmatrix} \Rightarrow \text{is-sorted}[\text{merged}[B, C]] \right).$$

■ A Degenerate Variant: Logical Algorithm Retrieval = Algorithm Verification

This variant of the logical algorithm retrieval problem is characterized by the fact that the knowledge base contains the inductive definitions of algorithm candidates without any further (proved) properties of the algorithms and we have to find out whether these candidates meet the given specification. In this variant, the algorithm retrieval problem is nothing else than an algorithm verification problem.

In our example, this means that we would find the inductive definition of three algorithms, say 'l', 'r', 'm', in the knowledge base, for example,

$$l[\langle\rangle] = \langle\rangle$$

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

$$\underset{x,y,\overline{z}}{\forall} (l[\langle x, y, \overline{z}\rangle] = x \frown l[\langle\overline{z}\rangle])$$

$$r[\langle\rangle] = \langle\rangle$$

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

$$\underset{x,y,z}{\forall} (r[\langle x, y, \overline{z}\rangle] = y \frown r[\langle\overline{z}\rangle])$$

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

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

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

$$\underset{x,\overline{x},y,\overline{y}}{\forall} \left(m[\langle x, \overline{x}\rangle, \langle y, \overline{y}\rangle] = \begin{Bmatrix} x \frown m[\langle\overline{x}\rangle, \langle y, \overline{y}\rangle] \Leftarrow x > y \\ y \frown m[\langle x, \overline{x}\rangle, \langle\overline{y}\rangle] \Leftarrow \neg\, x > y \end{Bmatrix} \right)$$

and we would have to prove that

$$\text{is-left-right-merge-structure}[l, r, m].$$

In many cases, such proofs can be carried out completely automatically with current mathematical software systems like *Theorema*. We do not give the details of the appropriate proof for our example because, in this paper, our main emphasis will be on the non-degenerate variant of the algorithm retrieval problem described in the next section. Note however that, typically, **such algorithm verification proofs are non-trivial inductive proofs**.

■ The Non-Degenerate Variant: Logical Algorithm Retrieval = Symbolic Computation Proving

■ The Problem

This variant of the logical algorithm retrieval problem is characterized by the fact that the knowledge base contains the inductive definitions of algorithm candidates together with a couple of (proved) properties of the algorithms that describe these algorithms "completely" and we have to find out whether these candidates meet the given specification. Here, the emphasis is on "completeness" of properties. We will expand on this notion in more detail below. In fact, this notion is the central notion in this paper. In this variant, the algorithm retrieval problem can be solved by "easy" proving, which more specifically is "symbolic computation proving" ("proving by rewriting", "high-school proving", "physicists' proving").

Before we go into an analysis of "complete knowledge" and "symbolic computation proving", we study the non-degenerate variant of the algorithm retrieval problem in our example:

We assume again that we find the inductive definition of three algorithms, say 'l', 'r', 'm', in the knowledge base, for example,

$$l[\langle\rangle] = \langle\rangle$$
$$\forall_{x} \; (l[\langle x\rangle] = \langle x\rangle)$$
$$\forall_{x,y,\bar{z}} \; (l[\langle x, y, \bar{z}\rangle] = x \frown l[\langle \bar{z}\rangle])$$
$$r[\langle\rangle] = \langle\rangle$$
$$\forall_{x} \; (r[\langle x\rangle] = \langle\rangle)$$
$$\forall_{x,y,\bar{z}} \; (r[\langle x, y, \bar{z}\rangle] = y \frown r[\langle \bar{z}\rangle])$$

$$m[\langle\rangle, \langle\rangle] = \langle\rangle$$
$$\forall_{y,\bar{y}} \; (m[\langle\rangle, \langle y, \bar{y}\rangle] = \langle y, \bar{y}\rangle)$$
$$\forall_{x,\bar{x}} \; (m[\langle x, \bar{x}\rangle, \langle\rangle] = \langle x, \bar{x}\rangle)$$
$$\forall_{x,\bar{x},y,\bar{y}} \left(m[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle] = \left\{ \begin{array}{l} x \frown m[\langle \bar{x}\rangle, \langle y, \bar{y}\rangle] \Leftarrow x > y \\ y \frown m[\langle x, \bar{x}\rangle, \langle \bar{y}\rangle] \Leftarrow \neg\, x > y \end{array} \right\} \right).$$

However, in addition, we also assume that the following (proved) properties is-left-right-structure[l,r] and is-merge-structure[m] were already available in the knowledge base, where is-left-right-structure and is-merge-structure are defined as follows:

$$\forall_{l,r}\left(\begin{array}{l}\text{is-left-right-structure}[l, r] \Leftrightarrow \\ \\ \\ \end{array}\right.$$

$$\left.\begin{array}{l}\forall_{\substack{\text{is-tuple}[X] \\ \neg\text{is-trivial-tuple}[X]}}\left(\begin{array}{l}l[X] \prec X \\ \text{is-tuple}[l[X]] \\ r[X] \prec X \\ \text{is-tuple}[r[X]] \\ (l[X] \asymp r[X]) \approx X\end{array}\right) \\ \\ \\ \end{array}\right)$$

$$\forall_{m}\left(\begin{array}{l}\text{is-merge-structure}[m] \Leftrightarrow \\ \\ \\ \end{array}\right.$$

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

where '$\asymp$' denotes concatenation.

We now, again, have to prove that

is-left-right-merge-structure[l, r, m].

Now, let's look to this proof:

■ The Proof

We assume

is-left-right-structure[l, r]

and

is-merge-structure[m]

and we have to prove

is-left-right-merge-structure[l, r, m],

i.e., for arbitrary X, Y, Z satisfying

is-tuple[X]
¬ is-trivial-tuple[X]
is-tuple[Y]
is-tuple[Z]

and

$l[X] \approx Y$
$r[X] \approx Z$
is-sorted[Y]
is-sorted[Z]

we have to prove

$$l[X] \prec X$$
$$\text{is-tuple}[l[X]]$$
$$r[X] \prec X$$
$$\text{is-tuple}[r[X]]$$
$$\text{is-tuple}[m[Y, Z]]$$
$$m[Y, Z] \approx X$$
$$\text{is-sorted}[m[Y, Z]].$$

Now, the first four goals are true by is-left-right-structure[l, r], the fifth goal is true by the assumptions and the first formula in is-merge-structure[m], and the seventh formula is true by the assumptions and the third formula in is-merge-structure[m].

Proof of the sixth goal:

$$m[Y, Z] \overset{(1)}{\approx} (Y \asymp Z) \overset{(2)}{\approx} (l[X] \asymp r[X]) \overset{(3)}{\approx} X$$

and, hence, by transitivity and symmetry of $\approx$ (see knowledge base in the appendix)

$$m[Y, Z] \approx X.$$

(1): By the assumptions and the second formula in is-merge-structure[m].

(2): By the assumptions and the property (see knowledge base in the appendix)

$$\forall_{\text{is-tuple}[A,B,Y,Z]} ((A \approx Y \wedge B \approx Z) \Rightarrow ((A \asymp B) \approx (Y \asymp Z))).$$

(3): By the last formula in is-left-right-structure [l, r].

■ Observation on the Proof Method and on Complete Knowledge Bases

Note that the above proof is neither trivial (i.e. just be renaming of bound variables and simple re-arrangement of the formulae) nor "difficult". More exactly, the proof is carried out exclusively by **"rewrite steps" ("symbolic computation steps", "high-school proving steps", "physicists' proving steps", "proving by algebraic manipulation")**. These are steps that involve only the "arbitrary but fixed" technique for universally bound variables in goals, substition of terms for universally bound variables in formulae in the knowledge base, replacement of a term by another term whose equality or congruence modulo an equivalence is already known, propositional steps including the expansion of formulae containing case distinction, and special inference rules for bounded quantifiers. This kind of proving is typical for proofs in high-school textbooks and also for "derivations" of formulae in applied mathematics, e.g. physics, where proofs normally do not involve alternating quantifiers ('$\forall \exists \forall$ ...' etc.) and, hence, no full-fledged predicate logic proving with the necessity of finding term instances for proving existential propositions, nor induction proofs for formulae universally quantified over inductive data types is necessary. Within "symbolic computation proving", the only available proof technique for universally quantified goals in symbolic computation proving is the "arbitrary but fixed" technique. "Symbolic computation proving" does not need any big ideas for the ingenious choice of suitable instances for existentially quantified proof goals, it has a purely computational flavor. The only ingenuity that may be necessary lies in finding the appropriate *sequence* of proof steps leading to a successful proof.

This notion of "symbolic computation proving", which is admittedly somewhat vague, is in some sense dual to the notion of a "**complete knowledge base**": We consider a knowledge base K to be *complete* if all formulae that are logical consequences of formulae in K and are not yet in K can be obtained from formulae in K by symbolic computation proving (i.e. "easy" proving). Conversely, if we had a clear understanding of what "complete" knowledge bases are, we would say that symbolic computation proving is the proving sufficient for

deriving all logical consequences from complete knowledge bases. Complete knowledge bases also could be (vaguely) characterized by saying that they admit a decision algorithm by symbolic computation.

Of course, by the fundamental theorems of logic (Goedel's incompleteness theorem etc.) there exist knowledge bases ("theories") that cannot be completed. However, practically, many of the hundreds of theories that arise in the layered build-up of mathematics *can* be completed in a quite concrete and natural way. Our strategic suggestion is that, in the systematic layered build-up of mathematics (including both nonalgorithmic and algorithmic mathematics), **after the introduction of any new, albeit intermediate and auxiliary concepts (operations, i.e. functions and predicates) by axioms or definitions, always complete the knowledge base for the new concepts by a systematic exploration of all possible interactions of the new concepts with all existing concepts and with themselves, before you proceed to introducing and exploring the next new concepts.** Ways for a systematic exploration and completion of the theories introduced by new concepts are described in [Buchberger 2000] and [Buchberger 2003b].

■ Another Degenerate Variant: Logical Algorithm Retrieval = Algorithm Invention

This variant of the logical algorithm retrieval problem is characterized by the fact that the knowledge base does not contain any algorithms that meet the given specification. In this case, the logical algorithm retrieval problem becomes the algorithm invention (algorithm synthesis) problem.

In our example, this means that we would have to synthesize algorithms 'l', 'r', 'm' satisfying

$$\text{is-left-right-merge-structure}[l, r, m]$$

from the operations available in the knowledge base. For example, the following algorithms could be synthesized

$$l[\langle\rangle] = \langle\rangle$$
$$\forall_{x} \; (l[\langle x\rangle] = \langle x\rangle)$$
$$\forall_{x,y,\bar{z}} \; (l[\langle x, y, \bar{z}\rangle] = x \frown l[\langle \bar{z}\rangle])$$
$$r[\langle\rangle] = \langle\rangle$$
$$\forall_{x} \; (r[\langle x\rangle] = \langle\rangle)$$
$$\forall_{x,y,\bar{z}} \; (r[\langle x, y, \bar{z}\rangle] = y \frown r[\langle \bar{z}\rangle])$$

$$m[\langle\rangle, \langle\rangle] = \langle\rangle$$
$$\forall_{y,\bar{y}} \; (m[\langle\rangle, \langle y, \bar{y}\rangle] = \langle y, \bar{y}\rangle)$$
$$\forall_{x,\bar{x}} \; (m[\langle x, \bar{x}\rangle, \langle\rangle] = \langle x, \bar{x}\rangle)$$
$$\forall_{x,\bar{x},y,\bar{y}} \left(m[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle] = \left\{ \begin{array}{l} x \frown m[\langle \bar{x}\rangle, \langle y, \bar{y}\rangle] \Leftarrow x > y \\ y \frown m[\langle x, \bar{x}\rangle, \langle \bar{y}\rangle] \Leftarrow \neg\, x > y \end{array} \right\} \right).$$

In fact, this synthesis is automatically possible by the method described in [Buchberger 2003a] and other algorithm synthesis methods. The method in [Buchberger 2003a] synthesizes the algorithms 'l', 'r', 'm' while establishing, in parallel, a proof of the fact that

$$\text{is-left-right-merge-structure}[l, r, m].$$

Note that, similarly to the algorithm verification problem, the algorithm synthesis problem requires to come up with **non-trivial inductive proofs** that go beyond the capability of "symbolic computation proving".

■ Decoupling Coupled Algorithm Specifications

■ Coupled and Decoupled Specifications

The relation between the two requirements (algorithm specifications)

is-left-right-merge-structure[l, r, m]

and

is-left-right-structure[l, r] ∧ is-merge-structure[m]

is a very interesting one: We call the second requirement "**decoupled**", whereas the first one is "**coupled**". This is an analogy, on the higher-order level for algorithms, to the notion of decoupled and coupled systems of, say, algebraic or differential equations.

Decoupling coupled requirements is of eminent practical importance: Typically, for given algorithms, the correctness proof of decoupled requirements is much easier than the correctness proof of coupled requirements. Analogously, the synthesis of algorithms satisfying decoupled requirements is much easier than the synthesis of algorithms satisfying coupled requirements. Thus, finding a decoupled requirement that entails a coupled requirement is an important subgoal in the exploration of theories. In our example, the decoupled requirement

is-left-right-structure[l, r] ∧ is-merge-structure[m]

entails the coupled requirement

is-left-right-merge-structure[l, r, m]

as we have shown in the section on algorithm retrieval by symbolic computation.

■ Explicit and Non-explicit Formulation of Specifications

Note furthermore that, in our example, the definition of both decoupled requirements

is-left-right-structure[l, r]

and

is-merge-structure[m]

are what we call "explicit requirement formulations" (or "explicit problem descriptions"): Formulae T (with the one free variable x) and P (with the two free variables x and y) are an explicit formulation of a requirement R for algorithms if

$$\forall_{f} \left(R[f] \Leftrightarrow \forall_{\substack{x \\ T[x]}} P[x, f[x]] \right)$$

(and analogously for algorithms with more than one argument and also algorithms with more than one output - which may also be conceived as two simultaneous algorithms). T, x, P, y are called the "input formula", "input variable", "input / output formula" (or, just, "output formula"), and "output variable" of the explicit requirement formulation, respectively.

(More exactly, the above formula should be written:

$$\forall_{f}\left(R[f] \Leftrightarrow \forall_{\substack{x \\ T}} P_{y \leftarrow f[x]}\right)$$

where '$\leftarrow$' denotes the substitution operation.)

For example, the requirement 'is-merge-structure' is explicitly defined by the input formula

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

with input variables 'Y' and 'Z' and the output formula

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

with the additional output variables 'M'.

Also, the requirement 'is-left-right-structure' is explicitly defined. For this we consider l and r as yielding two outputs for one input. The input formula is

$$\text{is-tuple}[X] \wedge \neg\, \text{is} - \text{trivial} - \text{tuple}[X]$$

with input variable 'X' and the output formula is

$$\begin{array}{l} L \prec X \\ \text{is-tuple}[L] \\ R \prec X \\ \text{is-tuple}[R] \\ (L \asymp R) \approx X \end{array}$$

with the additional *two* output variables 'L', 'R'.

Even the requirement 'is-left-rigth-merge-structure', although it is not decoupled, is explicitly defined by the formula in the introduction: We just look at the part

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

of the definition, which describes the interaction between 'left-split' and 'right-split' on the one side and 'merged' on the other side. (The other parts are explicit as we have seen above.) This is a formula that can be read as an explicit definition with input formula

$$\text{is-tuple}[X, Y, Z]$$

with input variables 'X', 'Y', 'Z' and output formula

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

with additional *three* output variables 'L', 'R', and 'M'.

Here is an example of an algorithm requirement (specification) that is inherently non-explicit, i.e. for which one cannot find an equivalent definition that is explicit:

$$\text{is-canonic-simplifier}[t, \sigma, \sim] \Leftrightarrow \begin{cases} \forall_{t[x]} \ (\sigma[x] \sim x) \\ \forall_{t[x,y]} \ (x \sim y \Rightarrow (\sigma[x] = \sigma[y])). \end{cases}$$

I have an easy proof of the fact
that this requirement cannot be made explict
but, unfortunately, time and space
does not permit me to give this proof
here.

(Additional remark: I do hope that,
in approximately 300 years from now,
somebody will find this paper and
try to find and write down a proof.)

More on coupled and decoupled, explicitly and non-explicitly defined requirements will be contained in [Buchberger 2003 b].

■ Conclusion

We have clarified the problem of "logical algorithm retrieval" and have emphasized the importance of two notions in this context:

- "symbolic computation proving" ("high-school proving", "physicists' proving", "algebraic proving", "table look-up proving", "basic proving")
- "complete knowledge base".

For the *Theorema* system, this has the following practical consequence: We will put decisive effort into implement a "basic prover" that can be used as an elementary building block for the problem of algorithm retrieval (and any other type of mathematical knowledge retrieval) and, at the same time, also as a sub-prover for all other, more sophisticated special provers for the various areas of mathematics.

■ References

[Asperti 2003]
Proceedings of the 2nd International Conference on Mathematical Knowledge Management,
Bologna, 2003.

[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 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. 2000] B. Buchberger, C. Dupre, Tudor Jebelean, F. Kriftner, Koji Nakagawa, D. Vasaru, W. Windsteiger.
The *Theorema* Project: A Progress Report.

In: 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), M. Kerber and M. Kohlhase (eds.), A.K. Peters, Natick, Massachusetts, pp. 98-113.

[Buchberger 2003a] B. Buchberger. 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.

[Buchberger 2003b] B. Buchberger.
Methods for Systematic Mathematical Theory Exploration.
Technical Report of the SFB (Special Research Area) Scientific Computing, October 2003, Johannes Kepler University, Linz, Austria.

[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.

■ Appendix: Knowledge Base for the Sorting Problem

■ Definitions

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

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

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

$$\forall_{x,y,\bar{z}} \left(\text{is-sorted}[\langle x, y, \bar{z} \rangle] \Leftrightarrow \left\{ \begin{array}{l} x \geq y \\ \text{is-sorted}[\langle y, \bar{z} \rangle] \end{array} \right. \right)$$

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

$$\forall_{y,\bar{y}} \left(\langle\rangle \not\approx \langle y, \bar{y} \rangle \right)$$

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

$$\forall_{x} (x \notin \langle\rangle)$$

$$\forall_{x,y,\bar{y}} \left((x \in \langle y, \bar{y} \rangle) \Leftrightarrow ((x = y) \vee x \in \langle \bar{y} \rangle) \right)$$

$$\forall_{a} (\text{dfo}[a, \langle\rangle] = \langle\rangle)$$

$$\forall_{a,x,\bar{x}} \left(\text{dfo}[a, \langle x, \bar{x} \rangle] = \left\{ \begin{array}{lll} \langle \bar{x} \rangle & \Leftarrow & x = a \\ x \smile \text{dfo}[a, \langle \bar{x} \rangle] & \Leftarrow & \text{otherwise} \end{array} \right. \right)$$

$$\forall_{\bar{y}} (\neg \langle\rangle \succ \langle \bar{y} \rangle)$$

$$\forall_{x,\bar{x}} (\langle x, \bar{x} \rangle \succ \langle\rangle)$$

$$\forall_{x,\bar{x},y,\bar{y}} \left(\langle x, \bar{x} \rangle \succ \langle y, \bar{y} \rangle \Leftrightarrow \langle \bar{x} \rangle \succ \langle \bar{y} \rangle \right)$$

$$\forall_{x,\overline{y}} \; (x \frown \langle \overline{y} \rangle = \langle x, \overline{y} \rangle)$$

$$\forall_{x,\overline{y}} \; (\langle \overline{y} \rangle \frown x = \langle \overline{y}, x \rangle)$$

$$\forall_{\overline{x},\overline{y}} \; (< \overline{x} > \; \asymp \; < \overline{y} > = \; < \overline{x}, \overline{y} >)$$

$$\forall_{X} \left(\text{is-tuple}[X] \iff \exists_{\overline{x}} \, (X = \langle \overline{x} \rangle) \right)$$

$$\forall_{X} \; (\text{is-empty-tuple}[X] \iff (X = \langle \rangle))$$

$$\forall_{X} \left(\text{is-singleton-tuple}[X] \iff \exists_{x} \, (X = \langle x \rangle) \right)$$

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

■ Axioms

$$\forall_{x,\overline{x},y,\overline{y}} \; ((\langle x, \overline{x} \rangle = \langle y, \overline{y} \rangle) \Leftrightarrow ((x = y) \wedge (\langle \overline{x} \rangle = \langle \overline{y} \rangle)))$$

$$\forall_{x,\overline{x}} \; (\langle x, \overline{x} \rangle \neq \langle \rangle)$$

■ Properties

$$\forall_{\text{is-trivial-tuple}[X]} \; \text{is-sorted}[X]$$

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

$$\forall_{\text{is-tuple}[X]} \; (X \approx X)$$

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

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

$$\forall_{\text{is-tuple}[A,B,Y,Z]} \; ((A \approx Y \wedge B \approx Z) \Rightarrow (A \asymp B) \approx (Y \asymp Z))$$

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

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