Algorithm Synthesis by Lazy Thinking: Using Problem Schemes

Bruno Buchberger[⋆] and Adrian Crăciun[⋆⋆]

Research Institute For Symbolic Computation,
Johannes Kepler University Linz,
A-4040 Linz, Austria
{bruno.buchberger, adrian.craciun}@risc.uni-linz.ac.at

Abstract. Recently, as part of a general formal (i.e. logic based) methodology for mathematical knowledge management we also introduced a method for the automated synthesis of correct algorithms, which we called the lazy thinking method. For a given concrete problem specification (in predicate logic), the method tries out various algorithm schemes and derives specifications for the subalgorithms in the algorithm scheme. In this paper, we show that the lazy thinking method can be significantly streamlined by applying it, in a preprocessing step, to problem schemes before we apply the result of the preprocessing to concrete problems.

1 Introduction to Algorithm Synthesis by Lazy Thinking

In this paper, we want to make a contribution to a particular subproblem of formal mathematical theory exploration (see [5]), namely the automated invention of (provenly correct) algorithms for given specifications. For this we recently introduced a new method, called "lazy thinking", see [3], which was quite successful in toy examples like the generation of sorting algorithms, see [7]. Very recently, we were able to show that the method is, in fact, quite powerful: In [4] we showed how the first author's Gröbner bases algorithm can be derived completely automatically from its specification by applying lazy thinking. We think that this is a major step forward, because the construction of Gröbner bases is a non-trivial problem, which was deemed to be outside the reach of automated algorithm synthesis.

In this paper we will streamline the lazy thinking methodology for algorithm invention by combining it with the idea of problem schemes. This makes the inventions of algorithms more efficient and, at the same time, the automatically

[⋆] Sponsored by FWF (Österreichischer Fonds zur Förderung der Wissenschaftlichen Forschung; Austrian Science Foundation), Project SFB 1302 ("Theorema") and by RICAM (Radon Institute for Computational and Applied Mathematics, Austrian Academy of Science, Linz). Currently visiting professor at Kyoto University, Graduate School of Informatics.

[⋆⋆] Supported by the Calculemus European Project and Project SFB 1302 ("Theorema").

generated correctness proofs will be better structured and easier to understand. This may also help to understand the lazy thinking method as a whole and to see its naturalness.

1.1 A Rough Description of the Lazy Thinking Synthesis Method

The two main ingredients of the lazy thinking method are:

- the use of *algorithm schemes*, and
- the *automated derivation of conjectures* for the specification of the subalgorithms from failing (automated) proof attempts for the correctness theorem.

An algorithm scheme is a predicate logic formula, that describes an algorithm (recursively) in terms of unspecified subalgorithms, together with the signature of the subalgorithms and a proof strategy to be used in the proof of the correctness theorem (the specification of the algorithm).

Roughly, the lazy thinking method proceeds as follows: We start from a formal (predicate logic) specification of the problem.

1. Try out, one after the other, various algorithm schemes stored in a library of algorithm schemes.
2. Attempt to prove the correctness theorem w.r.t. the given specification; this will typically fail because there is not enough knowledge about the subalgorithms.
3. Generate automatically the specification for one (or more) of the subalgorithms from the failing proof situation, add it to the knowledge base, and attempt the proof again.
4. Repeat the previous step in a recursive cascade, until the proof goes through; after the successful termination of the proof, the following will be true: Under the assumption that all the ingredient subalgorithms satisfy the specifications generated from the proof, the main algorithm satisfies the initial problem specification.
5. At this point, either, in the knowledge base at hand, algorithms are available that satisfy the specifications generated for the subalgorithms 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 specifications can be synthesized by another application of the same method in a next round of the procedure.

The degree of automation in the lazy thinking procedure depends on the degree of automation in the theorem provers available for the particular mathematical domain and the degree of automation in the procedure for obtaining requirements (specifications) for the subalgorithms from failing proof attempts. For the case studies on sorting, merging, and splitting tuples in [3, 7] both ingredients are completely automated in the frame of the Theorema system (see [6]) and, hence, the algorithm synthesis is completely automated. Surprisingly, the requirements generating algorithm described in [3] is also sufficient for the case study on Gröbner bases algorithm synthesis in [4]. However, the automated proving necessary for this case study is not yet fully implemented in Theorema although, in fact, the necessary automated theorem proving capability is less sophisticated, but technically more complex, than the one for the case studies on sorting. The implementation is under way and will be part of the PhD thesis of the second author.

1.2 Setting the Context

The Language. According to the view of theory exploration described in [5], one logic language frame (a version of higher order predicate logic) is sufficient for all aspects of mathematical theory exploration.

In particular, in this paper, we will use the Theorema version of predicate logic with sequence variables. The syntax should be self explanatory, for details see [6]. Here are some examples of the use of sequence variables:

$$\begin{aligned}&l[\langle\rangle] = 0,\\ &\underset{x,\bar{x}}{\forall}(l[\langle x, \bar{x}\rangle] = l[\langle\bar{x}\rangle] + 1).\end{aligned}$$

This is a recursive definition of the length of tuples. In Theorema, the tuple constructor is denoted by angle brackets. Hence, $\langle 2, 3, 2, 2, 1, 4\rangle$ is the ordered tuple (list) with the elements 2, 3, 2, 2, 1, 4; '$\bar{x}$' is a sequence variable. The recursive definition says that the length of the empty list is 0 and that the length of a list consisting of an object x and any finite sequence $\bar{x}$ of objects is the length of the list consisting of the finite sequence a plus 1. Hence,

$$l[\langle 2, 3, 1\rangle] = l[\langle 3, 1\rangle] + 1 = l[\langle 1\rangle] + 1 + 1 = l[\langle\rangle] + 1 + 1 + 1 = 0 + 3 = 3.$$

Note that, in the first step the sequence variable '$\bar{x}$' is bound to 3 *and* 1.

For a detailed formal account of predicate logic with sequence variables, see [17].

The programming language we use to express the algorithms is a restriction of the full Theorema language. In short, the programming language part of Theorema does not allow quantification over unbounded quantifiers. For more details, see [3].

Background Knowledge. Throughout this paper, we consider the context of general theory exploration. This means that when starting an exploration cycle, for instance the synthesis of an algorithm in a certain domain, knowledge about that domain will be available (possibly from previous exploration cycles).

We will present some case studies in the theory of tuples. Since we do not have enough space to present detailed definitions and properties in this paper, we will describe their meaning informally. For the exact definitions, refer to the Appendixes of [3, 7].

Problem Schemes. One of the most general problem schemes we can consider has the following form:

$$\underset{F,I,P}{\forall}(\mathit{construct\text{–}direct\text{–}solution}[F,I,P] \Leftrightarrow \underset{I[X]}{\forall} P[X,F[X]]),$$

which says that if the input to algorithm F satisfies the condition I, its output satisfies the (output) condition P w.r.t. its input. This output condition can in fact be a conjunction of various predicates that specify various desired properties for the algorithm F. The output condition can have particular forms, for instance it can describe a simplified version of the input, like in the following problem scheme:

$$\underset{F,I,P}{\forall}(\mathit{construct\text{–}simplifier}[F,I,P] \Leftrightarrow \underset{I[X]}{\forall} P[F[X]]).$$

In practice it is useful to store schemes in which the ingredients of the output condition are spelled out explicitly, like:

$$\underset{F,I,P,Q}{\forall}(\mathit{construct\text{–}direct\text{–}simplified\text{–}solution}[F,I,P,Q] \Leftrightarrow \bigwedge \begin{cases} \mathit{construct\text{–}simplifier}[F,I,Q] \\ \mathit{construct\text{–}direct\text{–}solution}[F,I,P] \end{cases}).$$

This gives hierarchical structure for the library of problem schemes. Both '*construct–simplifier*' and '*construct–direct–simplified–solution*' can be seen as instances of '*construct–direct–solution*'.

Remark. In many cases, it may also be desirable to specify the types of the input and output of problem specifications (and also the later algorithm schemes). In THEOREMA, this can be done by introducing appropriate additional unary predicates, see the examples below.

Algorithm Schemes. Here are some algorithm schemes for the domain of tuples:

$$\underset{F,e,g,h}{\forall}(\mathit{simple\text{-}tail\text{-}recursion\text{-}empty\text{-}general}[F,e,g,h] \Leftrightarrow \bigwedge \begin{cases} F[\langle\rangle] = e \\ \underset{x,\bar{y}}{\forall} F[\langle x,\bar{y}\rangle] = g[\langle x\rangle, F[\langle\bar{y}\rangle]] \end{cases}),$$

$$\underset{\substack{F,ee,eg,ge,\\ ggp,ggn,p}}{\forall}(\mathit{simple\text{-}merge\text{-}recursion\text{-}empty\text{-}general\text{–}binary}[F,ee,eg,ge,ggp,ggn,p] \Leftrightarrow$$
$$\bigwedge \begin{cases} F[\langle\rangle,\langle\rangle] = ee \\ \underset{x,\bar{y}}{\forall} F[\langle\rangle,\langle x,\bar{y}\rangle] = eg[\langle x,\bar{y}\rangle] \\ \underset{a,\bar{b}}{\forall} F[\langle a,\bar{b}\rangle,\langle\rangle] = ge[\langle a,\bar{b}\rangle] \\ \underset{a,\bar{b},x,\bar{y}}{\forall}(F[\langle a,\bar{b}\rangle,\langle x,\bar{y}\rangle] = \begin{cases} ggp[\langle a,\bar{b}\rangle,\langle x,\bar{y}\rangle,F[\langle a,\bar{b}\rangle,\langle\bar{y}\rangle]] \Leftarrow p[a,x] \\ ggn[\langle a,\bar{b}\rangle,\langle x,\bar{y}\rangle,F[\langle\bar{b}\rangle,\langle x,\bar{y}\rangle]] \Leftarrow \text{otherwise} \end{cases} \end{cases}),$$

$$\underset{F,e,g,h1,h2}{\forall}(\textit{double–recursion–empty–singleton–general}[F,e,g,h1,h2] \Leftrightarrow \bigwedge\begin{cases} F[\langle\rangle]=e \\ \underset{x}{\forall} F[\langle x\rangle]=s[\langle x\rangle] \\ \underset{x,\bar{y}}{\forall} F[\langle x,\bar{y}\rangle]=g[F[h1[\langle x,\bar{y}\rangle]],F[h2[\langle x,\bar{y}\rangle]]]\end{cases}),$$

which, in fact, is a special case of the domain-independent general algorithm scheme:

$$\underset{F,c,s,g,h1,h2}{\forall}(\textit{double–recursion}[F,c,s,g,h1,h2] \Leftrightarrow \underset{X}{\forall}(F[X]=\begin{cases} s[X] & \Leftarrow c[X] \\ g[F[h1[X]],F[h2[X]]] & \Leftarrow \text{otherwise}\end{cases})).$$

Note that the '*double-recursion*' scheme is also known as the '*divide-and-conquer*' scheme.

All of the above schemes are recursive. Since there is a strong connection between induction and recursion, a natural choice for the proof strategies associated to these schemes is some sort of induction, if the domain of the input variables is known to be inductive.

Additional conditions are attached to each scheme, such as domain preserving specifications (i.e. signatures of the unknown subalgorithms), and – specific to the '*divide–and–conquer*' scheme – the additional requirement that the preprocessing functions $h1$ and $h2$ make their input smaller, in a well–founded sense, than the output.

The Problem of Sorting. Consider the recursive definition of the unary predicate '*is-sorted*':

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

and the binary predicate '$\approx$', where $X \approx Y$ means that, in the tuples X and Y, the same elements occur the same number of times, see [3] for the detailed definition.

The problem specification for sorting in the theory of tuples is now obtained from the problem scheme '*construct-direct-simplified-solution*' by substituting *is–tuple*, *is-sorted* and $\approx$ for I, P and Q, respectively:

$$\textit{construct-direct-simplified-solution}[F, \textit{is–tuple}, \textit{is-sorted}, \approx],$$

which entails

$$\underset{\textit{is–tuple}[X]}{\forall}\bigwedge\begin{cases}\textit{is–sorted}[F[X]] \\ X\approx F[X]\end{cases}. \qquad \text{(Sort.P)}$$

As noted previously, the additional domain preserving specification

$$\underset{is\text{–}tuple[X]}{\forall} is\text{–}tuple[F[X]]$$

can be handled a priori and we assume this information every time. Note that in this case the output of the algorithm F has the same domain as its input.

We could now apply the lazy thinking method directly to this concrete problem for generating, automatically, a suitable algorithm F (by using any of the algorithm schemes above for F), like in [7]. *Rather, in this paper, we first apply the lazy thinking method to the general problem scheme construct-direct-solution*$[F,I,P]$. *Then we use the result of the lazy thinking method on the general problem scheme for any scheme that is an instance (such as construct–direct–simplified–solution) or concrete problem in this class (or in its instances), e.g. the sorting problem. This improves the efficiency of exploration enormously.*

2 Lazy Thinking Streamlined

2.1 Synthesis of a General Algorithm

We start from the problem scheme '*construct-direct-solution*', i.e. we want to find an algorithm F, whose output domain is described by the predicate O, such that

$$\underset{I[X]}{\forall} P[X, F[X]]. \qquad \text{(P)}$$

The formula (P) is called the correctness theorem or problem specification.

We use the algorithm scheme '*double–recursion*', i.e. the scheme

$$\underset{I[X]}{\forall} (F[X] = \begin{cases} s[X] & \Leftarrow c[X] \\ g[F[h1[X]], F[h2[X]]] & \Leftarrow \text{otherwise} \end{cases}). \qquad \text{(DC)}$$

Attached to the scheme are the signatures of the subalgorithms, which have the following form:

$$\underset{\substack{I[X]\\c[X]}}{\forall} O[s[X]], \qquad \text{(S.tr.dom)}$$

$$\underset{\substack{I[X]\\\neg c[X]}}{\forall} \bigwedge \begin{cases} I[h1[X]] \\ I[h2[X]] \end{cases}, \qquad \text{(Spl.ntr.dom)}$$

$$\underset{O[X_1,X_2]}{\forall} O[g[X_1, X_2]]. \qquad \text{(C.dom)}$$

We assume that I describes an inductive domain. Let $\prec$ be a well–founded ordering on this domain. Also attached to the algorithm scheme are the following conditions (specifications) on the splitting functions, $h1$ and $h2$:

$$\underset{\substack{I[X]\\\neg c[X]}}{\forall} \bigwedge \begin{cases} h1[X] \prec X \\ h2[X] \prec X \end{cases} \qquad \text{(Spl.ntr.ord)}$$

Now, for the proof of (P), we apply lazy thinking, i.e. we attempt the proof without knowing anything about the subalgorithms c, s, g, $h1$, and $h2$ in the algorithm scheme (except the conditions mentioned above). We use well–founded induction on $\prec$:

We take X_0 arbitrary but fixed such that $I[X_0]$ and assume as induction hypothesis:

$$\underset{\substack{I[X] \\ X \prec X_0}}{\forall} P[X, F[X]].$$

We try to prove

$$P[X_0, F[X_0]].$$

Case $c[X_0]$:

In this case, by the definition of F, it suffices to prove:

$$P[X_0, s[X_0]].$$

At this point, the proof is stuck because there is no specific knowledge available on s that could transform the proof situation further.

Failing Proof Analysis and Conjecture Generation. Applying the technique proposed in [3], we collect the current assumptions and the unproved goal in the formula

$$I[X_0] \wedge c[X_0] \Rightarrow P[X_0, s[X_0]],$$

and generalize it to

$$\underset{\substack{I[X] \\ c[x]}}{\forall} P[X, s[X]], \qquad \text{(Sp.tr)}$$

by the replacement:

$$X_0 \rightarrow X,$$

where 'X' is a new variable. This is now taken as a specification for s: It is clear that, if an s satisfies this specification, then the current goal can be proved from the current assumptions. Hence, we store this specification and continue with the proof.

Case $\neg c[X_0]$:

By the definition of F, it suffices to prove:

$$P[X_0, g[F[h1[X_0]], F[h2[X_0]]]].$$

From the domain signatures of F, and $h1$, $h2$ – (Spl.ntr.dom), we obtain:

$$O[F[h1[X_0]]],$$
$$O[F[h2[X_0]]],$$

By modus ponens, using (Spl.ntr.dom), (Spl.ntr.ord) and the well–founded induction hypothesis we obtain:

$$P[h1[X_0], F[h1[X_0]]],$$
$$P[h2[X_0], F[h2[X_0]]].$$

The proof is stuck, the proof situation cannot be transformed anymore.

Failing Proof Analysis and Conjecture Generation. Applying the technique proposed in [3], we collect the current assumptions and the unproved goal in the formula

$$\bigwedge \begin{cases} I[X_0] \\ \neg c[X_0] \\ O[F[h1[X_0]]] \\ O[F[h2[X_0]]] \\ P[h1[X_0], F[h1[X_0]]] \\ P[h2[X_0], F[h2[X_0]]] \end{cases} \Rightarrow P[X_0, g[F[h1[X_0]], F[h2[X_0]]]],$$

and generalize it to:

$$\underset{\substack{I[X], O[X_1, X_2] \\ \neg c[X]}}{\forall} \left(\bigwedge \begin{cases} P[h1[X], X_1] \\ P[h2[X]], X_2 \end{cases} \Rightarrow P[X, g[X_1, X_2]] \right), \qquad \text{(Sp.ntr)}$$

by first applying the replacements

$$F[h1[X_0]] \to X_1,$$
$$F[h2[X_0]] \to X_2,$$

and then $X_0 \to X$, where 'X', 'X_1', 'X_2' are new variables. This is now taken as a specification for g, $h1$, and $h2$: It is clear that, if functions g, $h1$, and $h2$ satisfy this specification, then the current goal can be proved from the current assumptions. Hence, we store this specification and, in fact, we are now done with the proof.

2.2 Summary of the Result

By the lazy thinking method for algorithm synthesis, we have automatically obtained the following theorem:

$$\underset{I[X]}{\forall} F[X] = \begin{cases} s[x] & \Leftarrow c[X] \\ g[F[h1[X]], F[h2[X]]] & \Leftarrow \text{otherwise} \end{cases}, \text{(DC)}$$

$$\bigwedge$$

$$\begin{array}{l}
\underset{\substack{I[X]\\c[X]}}{\forall} O[s[x]] \wedge \\
\underset{\substack{I[X]\\\neg c[X]}}{\forall} \bigwedge \left\{ \begin{array}{l} I[h1[X]] \\ I[h2[X]] \end{array} \right. \wedge \\
\underset{O[X_1,X_2]}{\forall} O[g[X_1,X_2]] \wedge \\
\underset{\substack{I[X]\\\neg c[X]}}{\forall} \bigwedge \left\{ \begin{array}{l} h1[X] \prec X \\ h2[X] \prec X \end{array} \right. \wedge \\
\underset{\substack{I[X]\\c[x]}}{\forall} P[X, s[X]] \wedge \\
\underset{\substack{I[X], O[X_1,X_2]\\\neg c[X]}}{\forall} \left(\bigwedge \left\{ \begin{array}{l} P[h1[X], X_1] \\ P[h2[X], X_2] \end{array} \right. \Rightarrow P[X, g[X_1, X_2]] \right)
\end{array} \qquad \text{(Specs)}$$

$$\Rightarrow$$

$$\underset{I[X]}{\forall} P[X, F[X]] \,. \qquad \text{(P)}$$

where (Specs) stands for the conjunction of (S.tr.dom), (Spl.ntr.dom), (C.dom), (Spl.ntr.ord), (Sp.tr) and (Sp.ntr), the specifications for the unknown subalgorithms in the recursive definition (DC).

This theorem can be read in two useful ways:

- Either we can consider the theorem as a kind of *preprocessing of the lazy thinking* approach for *synthesizing algorithms* for problems of the form: Find F such that (P) holds. In this interpretation, the theorem tells us that an algorithm F can be found by defining F by the formula (DC), with *any* subalgorithms s, g, $h1$, $h2$ and predicate c that satisfy the requirements (Specs). In this reading, we call the conditions (Specs) the *subalgorithm specifications* for the problem scheme (P) and the algorithm scheme (DC).
- Or, we consider the theorem as an *inference rule* for *verifying* that an algorithm of the form (DC) with *concrete operations* s, g, $h1$, $h2$ and c is correct with respect to the specification (P), for *concrete* predicates P, D. The conditions (Specs) are *sufficient conditions* for the partial correctness of the algorithm F.

2.3 Remarks

Partial Correctness vs. Total Correctness. For the above problem scheme and the above algorithm scheme we have proved that the algorithm is correct and terminates under the assumption that the auxiliary functions meet the specifications synthesized and the additional specifications associated to the algorithm scheme. Termination is ensured by the use of well–founded induction as the proof strategy. In general (due to the correspondence between induction and recursion), the use of (some sort of) induction (e.g. structural induction) to prove correctness of recursive schemes (e.g. the tail–recursion scheme) ensures termination.

For other algorithm schemes, termination may be a difficult problem that can only be handled by specific techniques for the specific auxiliary functions.

Problem Schemes Hierarchies. As pointed out in the previous section, '*construct–direct–simplified–solution*' can be seen as an instance of the '*construct–direct–solution*' problem scheme, (P). It is easy to specialize the result above for the '*construct–direct–simplified–solution*' problem scheme, by appropriately substituting the predicate P in (P) with a conjunction of predicates. Due to space limitations, we will not spell this out here, but will use the special form of the result in the subsequent case study.

3 Application to Concrete Algorithm Synthesis

3.1 The Case Studies

In this section, we will present a few synthesis case studies (sorting in the theory of tuples) which are not new (see [3, 7, 8]), and were carried out using the method of lazy thinking applied on concrete problems. We will mention the concrete problems and the algorithm schemes used in the respective case studies. We will then show how these problems and corresponding algorithm schemes can be seen as instances of the general synthesis problem presented in the previous section, which allows us to use directly the results of preprocessing lazy thinking, which provides the obvious advantage that we need not carry out several proofs/proof attempts.

3.2 Sorting of Tuples

We try to solve the sorting problem in the theory of tuples, i.e. find an algorithm F that satisfies (Sort. P), using the algorithm scheme:

$$\underset{is\text{-}tuple[X]}{\forall} F[X] = \begin{cases} s[X] & \Leftarrow is\text{–}trivial\text{–}tuple[X] \\ c[F[ls[X]], F[rs[X]]] & \Leftarrow \text{otherwise} \end{cases}, \qquad \text{(Sort.DC)}$$

with additional specifications

$$\underset{\substack{is\text{–}tuple[X] \\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall} \bigwedge \begin{cases} ls[X] \prec X \\ rs[X] \prec X \end{cases}. \qquad \text{(Sort.Sp.ntr.ord)}$$

and

$$\underset{\substack{is\text{–}tuple[X] \\ is\text{–}trivial\text{–}tuple[X]}}{\forall} is\text{–}tuple[s[X]], \qquad \text{(Sort.S.tr.dom)}$$

$$\underset{\substack{is\text{–}tuple[X] \\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall} \bigwedge \begin{cases} is\text{–}tuple[ls[X]] \\ is\text{–}tuple[rs[X]] \end{cases}, \qquad \text{(Sort.Spl.ntr.dom)}$$

$$\underset{is\text{–}tuple[X_1, X_2]}{\forall} is\text{–}tuple[c[X_1, X_2]]. \qquad \text{(Sort.C.dom)}$$

In the above, $\prec$ is the well–founded ordering on the domain of tuples, with the intuitive meaning 'has shorter length', '*is–tuple*' is the tuple detector, '*is–trivial–tuple*' describes either the empty tuple or singleton tuples (see [8] for the discussion on how to choose this predicate for the trivial case).

It is now near at hand to see that (Sort.P) and (Sort.DC) are instances of (P) and (DC), respectively, that '*is-tuple*' describes objects from an inductive domain with a well–founded ordering, $\prec$. We can now apply the preprocessed lazy thinking, taking also in account that (Sort.P) is in fact an instance of '*construct–direct-simplified–solution*', which yields the following specifications for s, ls, rs, c (we leave out the domain preserving specifications, since for the purpose of this presentation, they do not play any role in synthesis):

$$\underset{\substack{\textit{is-tuple}[X]\\ \textit{is–trivial–tuple}[X]}}{\forall} \bigwedge\begin{cases} \textit{is–sorted}[s[X]] \\ X \approx s[X] \end{cases}, \tag{Sort.Sp.tr}$$

$$\underset{\substack{\textit{is–tuple}[X]\\ \neg\textit{is–trivial–tuple}[X]}}{\forall} \bigwedge\begin{cases} ls[X] \prec X \\ rs[X] \prec X \end{cases}, \tag{Sort.Sp.ntr.ord}$$

$$\underset{\substack{\textit{is–tuple}[X,X_1,X_2]\\ \neg\textit{is–trivial–tuple}[X]}}{\forall} \left(\bigwedge\begin{cases} \textit{is–sorted}[X_1] \\ ls[X] \approx X_1 \\ \textit{is–sorted}[X_2] \\ rs[X] \approx X_2 \end{cases} \Rightarrow \bigwedge\begin{cases} \textit{is–sorted}[c[X_1, X_2]] \\ X \approx c[X_1, X_2] \end{cases} \right). \tag{Sort.Sp.ntr}$$

Now we see that the lazy thinking method applied to algorithms schemes and problem schemes and the subsequent application to the concrete problem instance yields the *natural specifications* for the subalgorithms and not just some contrived and artificial conditions without any intuitive interpretation: In particular, the specifications for c, ls, and rs are the natural specifications anybody would propose for *merging function* c and *splitting functions* ls and rs.

The specifications for the subalgorithms in the (Sort.DC) scheme were produced *directly*, by using the result of the preprocessing of the problem scheme (P) by lazy thinking with the algorithm scheme (DC), with the appropriate instantiations of their ingredients. When we applied the lazy thinking method to the concrete problem (Sort.P), with the algorithm scheme (Sort.DC), in Theorema, it took approximately 5 minutes runtime on a Pentium 4 machine to get the same result (see [7]).

If one now wants to go on with the synthesis of the subalgorithms for the concrete problem of sorting, it is essential to use the special knowledge available for the predicates '*is-sorted*' and $\approx$: For example, for synthesizing s, using all available knowledge on '*is-sorted*' and $\approx$, it turns out that the only possible function s satisfying the requirement is the identity.

For synthesizing c, ls, and rs, using available knowledge from the theory of tuples, it turns out that the above requirements can be replaced by the following (stronger) decoupled requirements (see [3] for details):

$$\underset{\substack{is\text{–}tuple[X]\\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall} \begin{cases} ls[X] \prec X \\ rs[X] \prec X \end{cases}, \tag{Sort.Spl.ord}$$

$$\underset{\substack{is\text{–}tuple[X]\\ \neg is\text{–}trivial\text{–}tuple[X]}}{\forall} (ls[X] \asymp rs[X]) \approx X, \tag{Sort.Spl}$$

$$\underset{is\text{–}tuple[X_1,X_2]}{\forall} \left(\begin{cases} is\text{–}sorted[X_1] \\ is\text{–}sorted[X_2] \end{cases} \Rightarrow \begin{cases} is\text{–}sorted[c[X_1,X_2]] \\ c[X_1,X_2] \approx (X_1 \asymp X_2) \end{cases} \right). \tag{Sort.Comb}$$

In the above, the first two are specifications for the splitting functions, and the last is the specification for merging. One can now go on and apply the same method for synthesizing algorithms for splitting and merging.

3.3 Sorting of Tuples, Continued: Splits

Starting with the problem given by the decoupled specifications for splits – (Sort.Spl.ord) and (Sort.Spl), after some analysis which yielded the definitions for the trivial cases, we showed in [8] that the following scheme can be used with the lazy thinking method, to synthesize the splitting algorithms (note that we use the explicit representation of tuples, i.e. in terms of the tuple constructor '⟨⟩'):

$$\underset{x,y}{\forall} \begin{cases} ls[\langle x,y\rangle] = \langle x\rangle \\ rs[\langle x,y\rangle] = \langle y\rangle \end{cases},$$
$$\underset{x,y,z}{\forall} \begin{cases} ls[\langle x,y,z\rangle] = \langle x,z\rangle \\ rs[\langle x,y,z\rangle] = \langle y\rangle \end{cases},$$
$$\underset{x,y,z,t,\bar{z}}{\forall} \begin{cases} ls[\langle x,y,z,t,\bar{z}\rangle] = cL[\langle x\rangle, ls[\langle z,t,\bar{z}\rangle]] \\ rs[\langle x,y,z,t,\bar{z}\rangle] = cR[\langle y\rangle, rs[\langle z,t,\bar{z}\rangle]] \end{cases}.$$

However, applying lazy thinking directly here has two disadvantages: (1) a lengthy, although not logically very deep, induction proof and (2) the need for a very particular structural induction rule, given by the structure of the recursive call (see [8]), meaning either that some analysis mechanism to produce this particular induction method must be available, or the user has to implement this particular scheme in the system. This example shows that the application of the lazy thinking method to concrete problems and algorithm schemes may lead to nontrivial issues concerning the theorem prover used to carry out the proofs.

Instead, by the observation (see details in [9]) that we can see the scheme above as an instance of the double-recursion scheme (DC) (with the preprocessing functions $h1[\langle x,y,z,t,\bar{z}\rangle] = \langle x,y\rangle$ and $h2[\langle x,y,z,t,\bar{z}\rangle] = \langle z,t,\bar{z}\rangle$, which clearly satisfy (Spl.ntr.ord) and (Spl.ntr.dom)), and the problem is a version of '*construct-direct-solution*', we can apply the preprocessed lazy thinking, and without the need for a proof, we get the specifications for the unknown subalgorithms cL and cR:

$$\underset{x,y,is\text{–}tuple[X,\ Y,\ Z]}{\forall}(\bigwedge\begin{cases}Y \prec X\\ Z \prec X\\ Y \asymp Z \approx X,\end{cases} \Rightarrow$$
$$\bigwedge\begin{cases}cL[\langle x\rangle, Y] \prec (\langle x,y\rangle \asymp X)\\ cR[\langle y\rangle, Z] \prec (\langle x,y\rangle \asymp X)\\ cL[\langle x\rangle, Y] \asymp cR[\langle y\rangle, Z] \approx (\langle x,y,\rangle \asymp X)\end{cases}).$$

These specifications are (not surprisingly) the same as in the case of the application of lazy thinking to the concrete problem and algorithm scheme. In [8] we show how given these specifications, we can retrieve [1] from the knowledge base that the concatenation function, $\asymp$, satisfies the specifications for cL and cR respectively, which means we are done with the synthesis of the splits.

3.4 Sorting of Tuples, Continued: Merging

Starting from the problem (Sort.Comb), we have showed in [7, 8] that we can using the algorithm scheme:

$$\begin{array}{l}
c[\langle\rangle, \langle\rangle] = cee,\\
\underset{y,\bar{y}}{\forall} c[\langle\rangle, \langle y, \bar{y}\rangle]] = ceg[y, \bar{y}],\\
\underset{x,\bar{x}}{\forall} c[\langle x, \bar{x}\rangle, \langle\rangle]] = cge[x, \bar{x}],\\
\underset{x,y,\bar{x},\bar{y}}{\forall} c[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle] = \begin{cases} cgg[\langle x\rangle, c[\langle \bar{x}\rangle, \langle y, \bar{y}\rangle]], \Leftarrow p[x,y]\\ cgg[\langle y\rangle, c[\langle x, \bar{x}\rangle, \langle \bar{y}\rangle]], \Leftarrow \neg p[x,y]\end{cases},
\end{array}$$

where cee, ceg, cge, cgg are subalgorithms of the scheme and p is an unknown predicate, we can synthesize the merging function:

$$\begin{array}{l}
c[\langle\rangle, \langle\rangle] = \langle\rangle,\\
\underset{y,\bar{y}}{\forall}(c[\langle\rangle, \langle y, \bar{y}\rangle] = \langle y, \bar{y}\rangle),\\
\underset{x,\bar{x}}{\forall}(c[\langle x, \bar{x}\rangle, \langle\rangle] = \langle x, \bar{x}\rangle),\\
\underset{x,\bar{x},y,\bar{y}}{\forall} c[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle] = \begin{cases} \langle x\rangle \asymp c[\langle \bar{x}\rangle, \langle y, \bar{y}\rangle] \Leftarrow x > y\\ \langle y\rangle \asymp c[\langle x, \bar{x}\rangle, \langle \bar{y}\rangle] \Leftarrow \neg x > y\end{cases},
\end{array}$$

where $\asymp$ is the concatenation in the theory of tuples.

Here we try to "cast" this problem scheme and algorithm scheme to match the general problem scheme (P) and general algorithm scheme (DC). The problem is that here we have a function with two arguments. Instead, we consider the arguments to be sequences of two tuples. In [9] we have shown that on this domain we can define a well–founded ordering, which makes it inductive.

Moreover, by taking

$$\underset{x,y,\bar{x},\bar{y}}{\forall} h1[bseq[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle]] = \begin{cases} bseq[\langle x\rangle, \langle\rangle] \Leftarrow x \geq y\\ bseq[\langle\rangle, \langle y\rangle] \Leftarrow \text{otherwise}\end{cases},$$
$$\underset{x,y,\bar{x},\bar{y}}{\forall} h2[bseq[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle]] = \begin{cases} bseq[\langle \bar{x}\rangle, \langle y, \bar{y}\rangle] \Leftarrow x \geq y\\ bseq[\langle x, \bar{x}\rangle, \langle \bar{y}\rangle] \Leftarrow \text{otherwise}\end{cases},$$

[1] It is here where the domain preservation specifications (signatures) play a role. However this is out of the scope of this paper. For details, see [8].

where *bseq* is a binary symbol that identifies the elements in the domain of binary sequences of tuples (the explicit representation of the objects in this domain), we can "cast" the above scheme into (DC). For details, see [9].

The specifications generated by the preprocessed lazy thinking in this case (translated back from the *bseq* formulation), for the subalgorithms *cee*, *ceg*, *cge*, *cgg*:

$$
\begin{array}{l}
\bigwedge\begin{cases}\text{is–sorted}[cee]\\ cee \approx \langle\rangle\end{cases},\\
\underset{y,\bar{y}}{\forall}\left(\text{is–sorted}[\langle y,\bar{y}\rangle] \Rightarrow \bigwedge\begin{cases}\text{is–sorted}[ceg[y,\bar{y}]]\\ ceg[y,\bar{y}] \approx \langle y,\bar{y}\rangle\end{cases}\right),\\
\underset{x,\bar{x}}{\forall}\left(\text{is–sorted}[\langle x,\bar{x}\rangle] \Rightarrow \bigwedge\begin{cases}\text{is–sorted}[cge[x,\bar{x}]]\\ cge[x,\bar{x}] \approx \langle x,\bar{x}\rangle\end{cases}\right),\\
\underset{x,y,\bar{x},\bar{y},\text{is–tuple}[Y]}{\forall}\left(\bigwedge\begin{cases}x \geq y\\ \text{is–sorted}[\langle x,\bar{x}\rangle]\\ \text{is–sorted}[\langle y,\bar{y}\rangle]\\ \text{is–sorted}[Y]\\ Y \approx \langle \bar{x},y,\bar{y}\rangle\end{cases} \Rightarrow \bigwedge\begin{cases}\text{is–sorted}[\langle x\rangle, Y]\\ cgg[\langle x\rangle, Y] \approx \langle x,\bar{x},y,\bar{y}\rangle\end{cases}\right),\\
\underset{x,y,\bar{x},\bar{y},\text{is–tuple}[Y]}{\forall}\left(\bigwedge\begin{cases}x < y\\ \text{is–sorted}[\langle x,\bar{x}\rangle]\\ \text{is–sorted}[\langle y,\bar{y}\rangle]\\ \text{is–sorted}[Y]\\ Y \approx \langle x,\bar{x},\bar{y}\rangle\end{cases} \Rightarrow \bigwedge\begin{cases}\text{is–sorted}[cgg[\langle x\rangle, Y]]\\ cgg[\langle y\rangle, Y] \approx \langle x,\bar{x},y,\bar{y}\rangle\end{cases}\right),
\end{array}
$$

lead to the merging algorithm presented above (see [8, 9] for complete details).

We have now completed the synthesis of the merge–sort algorithm, a solution for the problem (P).

3.5 Sorting of Tuples: Other Algorithms

As we already pointed out, the sorting problem (Sort.P) is an instantiation of the *construct–direct–simplified–solution* problem scheme:

$$
\underset{I[X]}{\forall}\bigwedge\begin{cases}P[F[X]]\\ Q[X,F[X]]\end{cases}.
$$

The preprocessing of lazy thinking can be applied to this problem scheme at some lower level, applying it to a scheme lower in the hierarchy of schemes, for instance the following, that depends on the domain of tuples (meaning that I is replaced with '*is–tuple*'):

$$
\underset{x,y,\bar{z}}{\forall}\begin{cases}F[\langle\rangle]\\ F[\langle x\rangle] = d[x]\\ F[\langle x,y,\bar{z}] = i[x, F[\langle y,\bar{z}\rangle]]\end{cases},
$$

with the attached proof strategy being structural induction on tuples.

We obtain completely automatically by the application of the lazy thinking method, the following subalgorithm specifications for the subalgorithms c, d, i:

$$\begin{array}{l} P[c], \\ Q[\langle\rangle, c], \\ \underset{x}{\forall}\left(\bigwedge\begin{cases} P[d[\langle x\rangle]] \\ Q[\langle x\rangle, d[\langle x\rangle]] \end{cases}\right), \\ \underset{\substack{x,y,\bar{z}, \\ is\text{–}tuple[Y]}}{\forall}\left(\bigwedge\begin{cases} P[Y] \\ Q[\langle y,\bar{z}\rangle, Y] \end{cases} \Rightarrow \bigwedge\begin{cases} P[i[x,Y]] \\ Q[\langle x,y,\bar{z}\rangle, i[x,Y]] \end{cases}\right). \end{array}$$

This example shows that sometimes it pays off to specialize a general scheme to a certain domain. Again, this suggests a hierarchy of algorithm schemes.

When we apply this to the problem of sorting, we obtain insertion sorting.

4 Comparison to Related Work

In this section we will briefly point out some of the similar approaches to algorithm synthesis and compare the similarities and differences between these and the lazy thinking approach (and in particular the preprocessing of lazy thinking). Then we will mention the connection to other related efforts in program verification.

Program synthesis is a well–studied field. Several approaches have been investigated, see [1] for an overview.

Constructive and deductive synthesis can be viewed from the same perspective: in both cases deduction is used to synthesize programs by solving unknowns during the applications of rules. In the context of proof planning in [15, 16], a proof of the correctness of an algorithm is set up, and the unknown parts of the algorithm (existential parts of the specification) are replaced with metavariables (middle–out reasoning), which will be instantiated as the proof planning progresses. Proof critics (see [13, 14]) are used to overcome failure of proofs and generate new lemmata. Lazy thinking is similar to this approach in that it also attempts to prove the correctness conjecture, and it uses the failure of the proof to generate conjecture and complete the proof. However, in the context of lazy thinking, conjectured lemmas are specifications of unknown subalgorithms in the schema. The use of schemas makes the proofs more efficient.

Algorithm schemas in the context of lazy thinking are similar to those in schema–based synthesis, with a template that captures the flow of the program and specifications (constraints) on the ingredients of the template.

In [11], a notion of generic correctness (modulo correctness of subalgorithms in the schema) of schemas – steadfastness – is used, and programs are synthesized by transforming steadfast schemas into correct programs. Similarly, preprocessing of lazy thinking ensures a notion of correctness of a schema (by the proof of the correctness theorem in the preprocessing of lazy thinking).

One of the most successful approaches to schema–based synthesis is that of D.R. Smith (see [20]), who uses a category theory framework to represent schemes and transformations. This setting ensures that transformations are correct. Moreover, a large library (hierarchy) of algorithm schemes is available and used to guide the synthesis.

Preprocessing lazy thinking gives a similar transformational flavor to our synthesis method. It allows to take offline some of the more difficult proof obligations – we apply once and for all lazy thinking and then all we need to do is show that the concrete problem and algorithm scheme selected are instances of a problem scheme and algorithm scheme that have been preprocessed.

All of the above approaches use some sort of declarative formalism: either functional programming (like in [20]) or logic programming (like [16, 11]). This is motivated by the goals of each approach. The aim of lazy thinking is to provide support for invention of new concepts (algorithms in this case) in the context of systematic theory exploration, see [5]. Therefore we use as the programming language (a restriction of) predicate logic. This allows us to remain in the same language frame throughout the synthesis process, and in all its aspects. The same is also achieved to some level (in the sense that algorithms and their specifications are expressed in the same language) in the logic programming setting (see [1] for a discussion), and in [12] a unified view of schemes and proof planning, which would allow natural integration of the proof tasks in the synthesis/transformation process is proposed.

Once an algorithm is synthesized in a certain language (in particular predicate logic), it can be transformed into an algorithm expressed in some other logic or formal system (see [18] for an overview of how this can be achieved).

There is a strong connection between verification and synthesis, as already pointed out in this paper. In fact the result presented here was motivated by the work of N. Popov, also in the frame of THEOREMA, see [19], who used Scott induction [10] (induction on the number of recursive calls) to derive sufficient verification conditions for the partial correctness of recursive functional algorithms of the form (DC), considered here. These are the same as those obtained by lazy thinking, (Specs). The difference between the two approaches is that here we allow only input from inductive domains, thus ensuring termination, while in [19] there is no restriction on the input, but the termination has to be proved separately.

5 Conclusions and Future Work

We have shown how the lazy thinking method can be significantly streamlined by applying it, in a preprocessing step, to problem schemes before we apply the result of the preprocessing to concrete problems. This saves a lot of proving effort and gives an easy structure to the resulting correctness proofs – from several minutes needed to produce proofs that turned out to be similar in structure to instant results using the preprocessed rules.

In particular, we have preprocessed a well–known algorithm scheme, divide and conquer. The algorithms synthesized by preprocessed lazy thinking using this scheme always terminate, due to the use of well–founded induction as a proof strategy for the correctness proof.

We have demonstrated the idea by a couple of case studies. In the ongoing PhD thesis of the second author under the guidance of the first author, an implementation of this approach is under way. In a first version of the implementation, we will provide a general lazy thinking module, which allows to synthesize the specification of the subalgorithms for arbitrary problem schemes and algorithms schemes in the area of tuple theory. A second version will then generalize the module to arbitrary inductive domains.

We will also study other algorithm schemes, such as critical–pair completion (CPC), first identified and described in [2], a quite general scheme, which captures common features of the Gröbner bases algorithm, Knuth–Bendix completion and resolution.

Additional properties of algorithms (as, for example, time and space complexity) may be formulated in a way which is similar to formulating other properties of algorithms. However some subtle points about language layers have to be taken into account, which we did not yet study completely.

References

1. D.Basin, Y.Deville, P. Flener, A. Hamfelt, J.F. Nilsson. Synthesis of Programs in Computational Logic. In M. Bruynooghe, K.–K. Lau (eds), *Program Development in Computational Logic*, Springer-Verlag, LNCS Series, **Vol. 3049**, June 2004, pp. 30–65.
2. B. Buchberger. History and Basic Features of The Critical–Pair/Completion Procedure. *Journal of Symbolic Computation*, **Vol. 3**, 1987, pp.3–38.
3. B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. In: D. Petcu, V. Negru, D. Zaharie, T. Jebelean (eds), *Proceedings of SYNASC 2003, 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing Timisoara, Romania, October 1 – 4, 2003*, Mirton Publishing, ISBN 973-661-104-3, pp. 2-26.
4. B. Buchberger. Towards the Automated Synthesis of a Gröbner Bases Algorithm. To appear in *RACSAM (Review of the Spanish Royal Academy of Sciences)*, 2004.
5. B. Buchberger. Algorithm–Supported Mathematical Theory Exploration: A Personal View and Strategy. In: J. Campbell (ed.), *Proceedings of AISC 2004 (7th Conference on Artifiicial Intelligence and Symbolic Computation, Sep. 22–24, 2004, Research Institute for Symbolic Computation, Hagenberg, Austria)*, Lecture Notes in Artificial Intelligence, Springer, to appear 2004.
6. B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger. The Theorema Project: A Progress Report. In: M. Kerber and M. Kohlhase (eds.), *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, August 6-7, 2000, St. Andrews, Scotland)*, A.K. Peters, Natick, Massachusetts, ISBN 1–56881–145–4, pp. 98–113.
7. B. Buchberger, A. Crăciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. in: F. Kamareddine (ed.), *Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, Nov. 25, 2003*, Electronic Notes

on Theoretical Computer Science, **Vol. 93**, `www.elsevier.com/locate/entcs`. Elsevier, ISBN 044451290X, 18 Feb. 2004, pp. 24-59.

8. A. Crăciun, B. Buchberger. Algorithm Synthesis Case Studies: Sorting of Tuples by Lazy Thinking. RISC Technical Report, 04–16, 2004.
9. A. Crăciun, B. Buchberger. Preprocessed Lazy Thinking: Synthesis of Sorting Algorithms. RISCTechnical Report, 04–17, 2004.
10. J. W. de Bakker, D. Scott. A Theory of Programs. IBM Seminar, Vienna, Austria, unpublished notes, 1969.
11. P. Flener, K.–K. Lau, M. Ornaghi, J.D. C. Richardson. An Abstract Formalization of Correct Schemas For Program Synthesis. *Journal of Symbolic Computation*, **30(1)**, 2000, pp. 93–127.
12. P. Flener, J. Richardson. A Unified View of Programming Schemas and Proof Methods. In A. Bossi (ed.), *Proceedings 9th International Workshop on Logic Program Synthesis and Transformation (LOPSTR'99)*, LNCS, **Vol. 1817**, Venezia, Italy, ISBN 3–540–67628–7, Springer–Verlag, 2000, pp.75–82.
13. A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov (ed.), *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence, **No. 624**, Springer–Verlag 1992, pp. 178–189.
14. A. Ireland, A. Bundy. Productive Use of Failure in Inductive Proof. *Special Issue of the Journal of Automated Reasoning*, **16(1–2)**, 1996, pp. 79–111.
15. I. Kraan. Logic Program Synthesis via Proof Planning. Phd. thesis, Department of Artificial Intelligence, University of Edinburgh, 1993.
16. I. Kraan, D. Basin, A. Bundy. Middle–out reasoning for Synthesis and Induction. *Journal of Automated Reasoning*, **16(1–2)**, 1996, pp. 113–145.
17. T. Kutsia, B. Buchberger, Predicate Logic with Sequence Variables and Sequence Function Symbols. To appear in A. Asperti, G.Bancerek, A. Trybulec (Eds.): *Proceedings of Mathematical Knowledge Management, Third International Conference, MKM 2004*, LNCS **3119**, Springer–Verlag.
18. J. Meseguer. Formal Interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence, Fort Laurerdale, Florida, January 1998*, url: `citeseer.ist.psu.edu/meseguer98formal.html`.
19. N. Popov. Verification of Simple Recursive Programs: Sufficient Conditions. RISC Technical Report 04–06, RISC Linz, Austria, 2004
20. D. R. Smith. Mechanizing the Development of Software. In M. Broy and R. Steinbrueggen (eds.), *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, IOS Press, ISBN 9051994591, Amsterdam, 1999, pp. 251–292.