# Algorithm Synthesis Case Studies: Sorting of Tuples by Lazy Thinking

Adrian Craciun, Bruno Buchberger
Research Institute for Symbolic Computation, Linz, Austria
{Adrian.Craciun, Bruno.Buchberger}@risc.uni-linz.ac.at

This report contains a detailed case study in algorithm synthesis, the synthesis of sorting algoritms for tuples. We apply the lazy thinking method for algorithm synthesis proposed by the second author in several cycles of exploration which will illustrate both the details of the application of the lazy thinking method as well as some aspects connected to the knowledge retrieval problem.

### ■ Disclaimer

This paper is an extension of a preliminary version of the [Buchberger, Craciun MKM 2003] paper, and it presents the case studies considered in more details, something that would not be possible in a conference paper. That is why some parts of this paper are similar or identical to the conference paper.

## ■ 1. Introduction: Lazy Thinking

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

- We start with a (predicate logic) specification of the algorithm in a particular data domain. We assume that all the auxiliary operations (functions and predicates) occurring in the specification are defined and all important properties of these operations are already known, i.e. that the "theory of these operations" is completely explored.

- The method now tries out, one after the other, various "algorithm schemes" that are stored in a library of algorithm schemes for the given mathematical domain. An algorithm scheme is a predicate logic formula that describes an algorithm (recursively) in terms of unspecified subalgorithms together with a proof method appropriate for (induction) proofs of properties of algorithms following this scheme.

- For the chosen algorithm scheme, the proof method is called for proving the correctness theorem, i.e. for proving that the unknown algorithm satisfies the given specification. Typically, this proof will fail because nothing is known about the unspecified subalgorithms.

- From the failing proof situation, by a conjecture generating algorithm, "requirements" (specifications) for the unknown subalgorithms are generated that enable the prover to complete the proof successfully. These requirements are added to the knowledge base and the proof of the correctness theorem is attempted again. Now, the proof will get over the failing situation and will either succeed or will fail again at some later proof situation.

- This procedure is iterated in a recursive cascade until the proof of the correctness theorem goes through (or one gives up). After successful termination, the following will be true: Under the assumption that all ingredient subalgorithms satisfy the requirements generated, the main algorithm satisfies the problem specification.

- In this stage, there are two possibilities: Either, in the initial knowledge base, algorithms are available that satisfy the requirements for the 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-supported systems* for formal mathematics and also for usage as a strategy for *human* algorithm invention and teaching. (In fact, the idea for the lazy thinking paradigm for algorithm synthesis arose while the first author was preparing a course on mathematical algorithm verification for high-school teachers in October 2001.)

Naturally, there are connections to *formal methods*, (automated) *software engineering*. For instance, the idea of capturing the accumulated knowledge of algorithm development in software patterns, collecting them in pattern catalogues is similar to the one of algorithm schemes in a library. For an overview and further references, see [Schmid,Johnson,Fayad 1996]. However, these are directed towards specific object–oriented or concurrent, parallel and distributed programming languages.

The lazy thinking method however, is a method of synthesis in *computational logic*, similar to the ones described in [Basin et al. 2003]. Some of the ideas in lazy thinking are similar to previous work in this domain (and are highlighted in the following). The deductive synthesis method, of the type described in [Kraan 1993], [Lacey 1999], [Armando 1999] use proof planning to set up the proof of the correctness of an algorithm, replaces the unknown parts of the algorithm (existential parts of the specification) with metavariables which will be instantiated as the proof planning progresses. The proof plan roughly consists of the application of induction, followed by rippling (a metalevel method to guide rewriting) and

Similar ideas were introduced in [Armando 1999], [Basin et al. 2003], [Flener 1995], [Merrit, Lau 1997], and [Smith 1985]. However, in one respect or the other, our synthesis method is different from each of those.

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

- The proof objects generated by the automated theorem proving part of the system must be open for post-processing. In other words, black-box provers are not suitable for the method because the important (automated) requirement generation for subalgorithms is based on a detailed analysis of the proof objects generated.

- For the same reason, it is also crucial that the automated theorem provers used in the system yield proof objects also in the case that proofs fail. In distinction to other algorithm synthesis methods, which extract algorithms from *successful* proofs, the essence of our method is the automated generation of requirements for subalgorithms from *failing* proofs!

- The automated theorem provers used in the system should be "natural style" provers, e.g. natural deduction provers or variants thereof: The generation of requirements for the unknown subalgorithms is crucially based on the analysis of the temporary assumptions and temporary proof goals in the failing proof object.

In this paper, all our formal developments will be given in the *Theorema* system, see [Buchberger et al. 1997, 2000]. In particular, all formulae will be given in the *Theorema* syntax. Also, the implementation of the method is described in the concrete example of the *Theorema* system.

# ■ 2. Synthesis of the Sorting Algorithm, Specifications for Subalgorithms

## ■ 2.1. Problem Specification

We want to synthesize an algorithm 'sorted' that meets the following specification

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

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'):

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

and

$$\begin{array}{l} \langle\rangle \approx \langle\rangle \\ \forall_{y,\bar{y}} \ \langle\rangle \not\approx \langle y, \bar{y} \rangle \\ \forall_{x,\bar{x},\bar{y}} \ (\langle x, \bar{x} \rangle \approx \langle \bar{y} \rangle \Leftrightarrow (x \in \langle \bar{y} \rangle \wedge \langle \bar{x} \rangle \approx \text{dfo}[x, \langle \bar{y} \rangle])). \end{array}$$

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 auxiliary functions until we arrive at the basic operations on tuples. The definitions of all these auxiliary operations and also the formulae describing various properties of these auxiliary operations are supposed to be contained in the knowledge base.

## ■ 2.2. Algorithm Scheme

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

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

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

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

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

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

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

$$\forall_{\substack{\text{is–tuple[X]} \\ \text{is–trivial–tuple[X]}}} \text{is–tuple[special[X]]}$$

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

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

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

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

as a part of the recursive definition, which guarantees termination of the algorithm in case '≺' is a Noetherian predicate. (In our case, we will use the predicate 'has shorter length' for '≺', 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, $\forall_{\text{is–tuple}[X]} A[X]$ 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 $\forall_{\substack{\text{is–tuple}[Y] \\ Y \prec \langle \overline{x0} \rangle}} A[Y]$.

This particular induction method 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 schemes libraries" that can then be used, in the way which we demonstrate in this paper, for inventing concrete algorithms for concrete problems.

## ■ 2.3. Algorithm Synthesis by Lazy Thinking: First Round

We now start from the following situation:

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

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

We do now the following:

- We include the algorithm scheme for 'sorted', the type requirements for the auxiliary functions, and the requirements on the decreasing length of 'left–split' and 'right–split' into the knowledge base.

- Then we start attempting to prove the correctness theorem

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

- Of course, this proof will not succeed because, at this moment, essentially nothing is known about the auxiliary functions 'merged', 'left-split' etc. We proceed with the proof until the proof gets stuck.

- When it got stuck, we analyze the current, failing proof situation and try to conjecture requirements (properties) of the auxiliary subalgorithms 'special', 'left–split', 'right-split', 'merged' that would make it possible to get over the failing proof situation. (It is essential for the automation of the method that this step of conjecturing suitable requirements can be automated!)

- We add the conjectured requirements to the knowledge base and repeat the whole process, i.e. we go to the next round in the algorithm invention process.

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

*Proof Attempt Begin*

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

We assume

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

and the induction hypothesis

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

and we show

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

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

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

In this case, we have to show

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

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

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

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

$$\text{is–sorted}[\text{special}[\langle \overline{x} \overline{o} \rangle]]. \quad \text{(G3)}$$

For (G2), by the fact that

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

it suffices to prove that

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

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

*Proof Attempt End*

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

Now we analyze the failing proof situation and find:

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

is–trivial–tuple[⟨x̄ō⟩].

○ We have the temporary goal:

special[⟨x̄ō⟩] = ⟨x̄ō⟩.

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'

$$\forall_{\text{is–trivia–tuple[X]}} (\text{special[X]} = \text{X})$$

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

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

## ■ 2.4. Algorithm Synthesis by Lazy Thinking: Second Round

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

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

*Proof Attempt Begin*

CASE ¬is–trivial–tuple[⟨x̄ō⟩]:

In this case, we have to show

is–sorted–version[⟨x̄ō⟩, merged[sorted[left–split[⟨x̄ō⟩], sorted[right–split[⟨x̄ō⟩]]]]].

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

(H1) is–tuple[merged[sorted[left–split[⟨x̄ō⟩], sorted[right–split[⟨x̄ō⟩]]]]],

(H2) ⟨x̄ō⟩ ≈ merged[sorted[left–split[⟨x̄ō⟩], sorted[right–split[⟨x̄ō⟩]]]],

(H3) is–sorted[merged[sorted[left–split[⟨x̄ō⟩], sorted[right–split[⟨x̄ō⟩]]]]].

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

is–sorted–version[left–split[⟨x̄ō⟩], sorted[left–split[⟨x̄ō⟩]]],

is–sorted–version[right–split[⟨x̄ō⟩], sorted[right–split[⟨x̄ō⟩]]].

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

(AL1) is–tuple[sorted[left–split[⟨x̄ō⟩]]],

(AL2) left–split[⟨x̄ō⟩] ≈ sorted[left–split[⟨x̄ō⟩]],

(AL3) is–sorted[sorted[left–split[⟨x̄ō⟩]]],

(AR1) is–tuple[sorted[right–split[⟨x̄ō⟩]]],

(AR2) right–split[⟨x̄ō⟩] ≈ sorted[right–split[⟨x̄ō⟩]],

(AR3) is–sorted[sorted[right–split[⟨x̄ō⟩]]].

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

We cannot prove (H2) and (H3).

*Proof Attempt End*

Now we analyze the failing proof situation and find:

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

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

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

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

## ■ 2.5. The Result of the Synthesis

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

$$\begin{aligned} &\text{Knowledge[is–sorted–version]} \\ &\Rightarrow \\ &\forall_{\text{special,merged,left–split,right–split,sorted}} \\ &\quad \Big( \text{Is–Merge–Sort–Algorithm[sorted, special, merged, left–split, right–split]} \\ &\qquad \Rightarrow \forall_{\text{is–tuple[X]}} \text{is–sorted–version[X, sorted[X]]} \Big) \end{aligned}$$

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:

$$\forall_{\text{special,merged,left–split,right–split,sorted}} \text{Is–Merge–Sort–Algorithm[special, merged, left–split, right–split, sorted]}$$

$$\Leftrightarrow$$

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

The theorem says that,

**if**

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

**then**

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

The most important and most interesting parts of this theorem are the two requirements stating that the function 'merged' preserves sortedness and elements. These two requirements are exactly what people would naturally consider as the characteristic properties of merging. The amazing phenomenon is that exactly these two requirements are invented completely automatically, without any prior intuition or semantic understanding, by our "lazy thinking"

method. In fact, the exact formulation of the requirements invented by our method, are slightly more general than the requirements one would expect naturally. This is, of course, good because the weaker the requirements the more functions 'merged', 'left–split', and 'right–split' satisfy the requirements!

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

In [Buchberger 2003a] we go a step further and analyze the knowledge base from which we started with the objective to find out those "minimal" properties of the ingredient operations that suffice for proving the relative correctness theorem. We do not describe this idea in the present paper.

## ■ 2.6. Mathematical Knowledge Retrieval

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

Rather, we are faced with the following problem:

- Given a knowledge base K, operation names f, ..., 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, knowledge retrieval in our context is essentially a proving problem!

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

then "finding" operations in K that satisfy the requirement consists in trying out all possible triples of functions l, r, m that occur in K and finding out whether the requirement R[ l, r, m] can be proved from the formulae in the knowledge base. In our case, in particular, one could try L, R, M and try to prove that R[ L, R, M] holds. One sees that this task is nothing else than proving that the algorithms L, R, M are correct w.r.t. to the specification R[ L, R, M]. Of course, such proofs, may be difficult, moderately difficult, easy, or trivial depending on how much knowledge is available in the knowledge base. We have explained this in big detail in a recent technical report, see [Buchberger 2003b]. In particular, if no suitable functions l, r, m are available in the knowledge base, then the problem becomes another synthesis problem, namely the problem of synthesizing algorithms l, r, m satisfying the requirement R[ l, r, m]. We will solve this problem, again by our lazy thinking algorithm synthesis method in the next section.

# ■ 3. Synthesis of the Merging Function

## ■ 3.1. Problem Specification

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

$$\forall_{\substack{\text{is–tuple}[X] \\ \neg\text{is–trivial–tuple}[X]}} \begin{cases} L[X] \prec X \\ \text{is–tuple}[L[X]] \\ R[X] \prec X \\ \text{is–tuple}[R[X]] \end{cases}$$

$$\forall_{\text{is–tuple}[Y,Z]} \text{is–tuple}[M[Y, Z]]$$

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

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

$$\forall_{\substack{\text{is–tuple}[X] \\ \neg\text{is–trivial–tuple}[X]}} \begin{cases} L[X] \prec X \\ \text{is–tuple}[L[X]] \\ R[X] < X \\ \text{is–tuple}[R[X]] \end{cases}$$

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

$$\forall_{\text{is–tuple}[Y,Z]} \text{is–tuple}[M[Y, Z]]$$

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

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

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

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

$$\forall_{\text{is–tuple}[Y,Z]} \text{is–tuple}[c[Y, Z]]$$

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

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

## ■ 3.2. Algorithm Scheme

$$c[\langle\rangle, \langle\rangle] = cee$$
$$\forall_{y,\bar{y}} c[\langle\rangle, \langle y, \bar{y}\rangle] = ceg[y, \bar{y}]$$
$$\forall_{x,\bar{x}} c[\langle x, \bar{x}\rangle, \langle\rangle] = cge[x, \bar{x}]$$

Now for the general case:

$$\forall_{x,\bar{x},y,\bar{y}} c[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle] = \left\{ \begin{array}{l} cgg1[x, c[\langle\bar{x}\rangle, \langle y, \bar{y}\rangle]] \Leftarrow p[x, y] \\ cgg2[y, c[\langle x, \bar{x}\rangle, \langle\bar{y}\rangle]] \Leftarrow \neg\, p[x, y] \end{array} \right\}$$

The type requirements are the following:

$$\text{is–tuple}[cee]$$
$$\forall_{x,\bar{x}} \text{is–tuple}[ceg[x, \bar{x}]]$$
$$\forall_{x,\bar{x}} \text{is–tuple}[cge[x, \bar{x}]]$$
$$\forall_{x,\text{is–tuple}[X]} \left\{ \begin{array}{l} \text{is–tuple}[cgg1[x, X]] \\ \text{is–tuple}[cgg2[x, X]] \end{array} \right\}$$

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

The corresponding inductive proof technique is:

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

1. Prove A[⟨⟩].

2. Take $x_o$ and $\overline{x_0}$ arbitrary but fixed, assume A[⟨$\overline{x_0}$⟩] and prove A[⟨$x_0$, $\overline{x_0}$⟩].

This proof technique can be expanded to the following proof technique for conditioned formulae:

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

1. Assume C[⟨⟩] and show B[⟨⟩].

2. Take $x_o$ and $\overline{x_0}$ arbitrary but fixed. Assume ¬ C[⟨$\overline{x_0}$⟩] and C[⟨$x_0$, $\overline{x_0}$⟩] and prove B[⟨$x_0$, $\overline{x_0}$⟩].

3. Now assume C[⟨$\overline{x_0}$⟩], B[⟨$\overline{x_0}$⟩], and C[⟨$x_0$, $\overline{x_0}$⟩] and prove B[⟨$x_0$, $\overline{x_0}$⟩].

Note that 2. and 3. are appropriate because the assumption C[⟨$\overline{x_0}$⟩]⇒B[⟨$\overline{x_0}$⟩] is equivalent to ¬C[⟨$\overline{x_0}$⟩] ∨ (C[⟨$\overline{x_0}$⟩] ∧ B[⟨$\overline{x_0}$⟩]) and, hence, the proof splits into the two cases ¬C[⟨$\overline{x_0}$⟩] and C[⟨$\overline{x_0}$⟩] ∧ B[⟨$\overline{x_0}$⟩].

## ■ 3.3. Algorithm Synthesis by Lazy Thinking: First Round

We first prove

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

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

Now we try to prove

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

by induction for formulae with conditions.

**Induction Base for Y**: Assume is–sorted[⟨⟩] . Prove $\underset{\substack{\text{is–tuple[Z]}\\ \text{is–sorted[Z]}}}{\forall} \left\{ \begin{matrix} \langle\rangle \asymp \text{Z} \approx \text{c[⟨⟩, Z]} \\ \text{is–sorted[c[⟨⟩, Z]]} \end{matrix} \right\}$

**Induction Base for Y, Induction Base for Z**: Assume is–sorted[⟨⟩]. Prove $\left\{ \begin{matrix} \langle\rangle \asymp \langle\rangle \approx \text{c[⟨⟩, ⟨⟩]} \\ \text{is–sorted[c[⟨⟩, ⟨⟩]]} \end{matrix} \right\}$

Now,

$$\langle\rangle \asymp \langle\rangle \approx \text{c[⟨⟩, ⟨⟩]} \leftrightarrow \langle\rangle \approx \text{cee}.$$

This proof fails but straightforwardly leads to the requirement

$$\text{cee} = \langle\rangle.$$

Furthermore, using already this requirement,

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

**Induction Base for Y, Induction Step for Z, First Case**: Assume $\neg$ is–sorted$[\langle \overline{z_0} \rangle]$, is–sorted$[\langle z_0, \overline{z_0} \rangle]$. Prove $\langle\rangle \asymp \langle z_0, \overline{z_0} \rangle \approx \text{c}[\langle\rangle, \langle z_0, \overline{z_0} \rangle]$ and is–sorted$[\text{c}[\langle\rangle, \langle z_0, \overline{z_0} \rangle]]$.

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

**Induction Base for Y, Induction Step for Z, Second Case**: Assume is–sorted$[\langle \overline{z_0} \rangle]$, $\langle\rangle \asymp \langle \overline{z_0} \rangle \approx \text{c}[\langle\rangle, \langle \overline{z_0} \rangle]$, is–sorted$[\text{c}[\langle\rangle, \langle \overline{z_0} \rangle]]$,

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

*Case* $\overline{z_0} =$ *:* (the empty sequence)

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

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

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

Furthermore,

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

This proof fails but (automatically) leads to the requirement

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

## ■ 3.4. Algorithm Synthesis by Lazy Thinking: Second Round

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

**Induction Base for Y, Induction Step for Z, Second Case**: Assume is–sorted$[\langle \overline{z_0} \rangle]$, $\langle\rangle \asymp \langle \overline{z_0} \rangle \approx \text{c}[\langle\rangle, \langle \overline{z_0} \rangle]$, is–sorted$[\text{c}[\langle\rangle, \langle \overline{z_0} \rangle]]$,

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

*Case* $\overline{z_0} = z_1, \overline{z}$ (for certain $z_1, \overline{z}$):

From the assumptions we know:

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

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

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

Now,

$$\langle\rangle \asymp \langle z_0, \overline{z_0}\rangle \approx c[\langle\rangle, \langle z_0, \overline{z_0}\rangle] \leftrightarrow \langle z_0, z_1, \overline{z}\rangle \approx \text{ceg}[z_0, z_1, \overline{z}] \leftrightarrow$$
$$(z_0 \in \text{ceg}[z_0, z_1, \overline{z}] \wedge \langle z_1, \overline{z}\rangle \approx \text{dfo}[z_0, \text{ceg}[z_0, z_1, \overline{z}]]) \leftarrow (z_0 \in \text{ceg}[z_0, z_1, \overline{z}] \wedge (\text{ceg}[z_1, \overline{z}] \approx \text{dfo}[z_0, \text{ceg}[z_0, z_1, \overline{z}]])).$$

This proof fails but leads to the requirement

$$\forall_{z0,z1,\overline{z}} (z0 \in \text{ceg}[z0, z1, \overline{z}] \wedge (\text{ceg}[z1, \overline{z}] \approx \text{dfo}[z0, \text{ceg}[z0, z1, \overline{z}]]))$$

Furthermore,

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

This proof fails but leads to the requirement

$$\forall_{z0,z1,\overline{z}} \left(\begin{Bmatrix} \text{is–sorted}[\text{ceg}[z1, \overline{z}]] \\ z0 \geq z1 \end{Bmatrix} \Rightarrow \text{is–sorted}[\text{ceg}[z0, z1, \overline{z}]]\right)$$

## ■ 3.5. Algorithm Synthesis by Lazy Thinking: Third Round

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

**Induction Step for Y, First Case:**

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

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

Assume

$$\text{is–sorted}[\langle\overline{y_0}\rangle]$$

$$\begin{Bmatrix} \langle\overline{y_0}\rangle \asymp \langle\rangle \approx c[\langle\overline{y_0}\rangle, \langle\rangle] \\ \text{is–sorted}[c[\langle\overline{y_0}\rangle, \langle\rangle]] \end{Bmatrix}$$

$$\text{is–sorted}[\langle y_0, \overline{y_0}\rangle].$$

Prove

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

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

$$\forall_{y} (\text{is–sorted}[\text{cge}[y]])$$

$$\forall_{y0,y1,\overline{y}} (y0 \in \text{cge}[y0, y1, \overline{y}] \wedge (\text{cge}[y1, \overline{y}] \approx \text{dfo}[y0, \text{cge}[y0, y1, \overline{y}]]))$$

$$\forall_{y0,y1,\overline{y}} \left( \begin{cases} \text{is–sorted}[\text{cge}[y1, \overline{y}]] \\ y0 \geq y1 \end{cases} \Rightarrow \text{is–sorted}[\text{cge}[y0, y1, \overline{y}]] \right)$$

## ■ 3.6. Algorithm Synthesis by Lazy Thinking: Fourth Round

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

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

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

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

Assume

$$\text{is–sorted}[\langle \overline{z_0} \rangle]$$

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

$$\text{is–sorted}[c[\langle y_0, \overline{y_0} \rangle, \langle \overline{z_0} \rangle]]$$

$$\text{is–sorted}[\langle z_0, \overline{z_0} \rangle]$$

Prove

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

and

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

Now, we distinguish two cases.

**Case p[$y_0$,$z_0$]:**

In this case,

$$\langle y_0, \overline{y_0}, z_0, \overline{z_0} \rangle \approx c[\langle y_0, \overline{y_0} \rangle, \langle z_0, \overline{z_0} \rangle] \leftrightarrow \langle y_0, \overline{y_0}, z_0, \overline{z_0} \rangle \approx \text{cgg1}[y_0, c[\langle \overline{y_0} \rangle, \langle z_0, \overline{z_0} \rangle]] \quad (*)$$

By the induction hypothesis on Y

$$\forall_{\substack{\text{is–tuple}[Z] \\ \text{is–sorted}[Z]}} \left\{ \begin{matrix} \langle \overline{y_0} \rangle \asymp Z \approx c[\langle \overline{y_0} \rangle, Z] \\ \text{is–sorted}[c[\langle \overline{y_0} \rangle, Z]] \end{matrix} \right\}$$

we know, in particular,

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

We are now stuck at the proof of (*).

By the requirement generation algorithm, we can get the requirements

$$\forall_{x,\text{is–tuple}[Y,Z]} (Y \approx Z \Rightarrow \text{cgg1}[x, Y] \approx \text{cgg1}[x, Z])$$

$$\forall_{y,\bar{y},z,\bar{z}} \langle y, \bar{y}, z, \bar{z}\rangle \approx \text{cgg1}[y, \langle \bar{y}, z, \bar{z}\rangle]$$

and by adding them to the knowledge base, we can prove (*):

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

Furthermore,

$$\text{is–sorted}[c[\langle y_0, \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]] \leftrightarrow \text{is–sorted}[\text{cgg1}[y_0, c[\langle \bar{y_0}\rangle, \langle z_0, \bar{z_0}\rangle]]].$$

We already know, by the induction hypothesis on Y:

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

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

and we also know

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

We are now stuck at the proof of (**).

However, with the (automatically generated) requirement

$$\forall_{y,\bar{y},z,\bar{z},\text{is–tuple}[Y]} \left( \bigwedge \left\{ \begin{array}{l} \text{is–sorted}[Y] \\ Y \approx \langle \bar{y}, z, \bar{z}\rangle \\ \text{is–sorted}[\langle y, \bar{y}\rangle] \end{array} \right\} \Rightarrow \text{is–sorted}[\text{cgg1}[y, Y]] \wedge \left\{ \begin{array}{l} \text{is–sorted}[\langle z, \bar{z}\rangle] \\ p[y, z] \end{array} \right. \right)$$

we can successfully get over this proof situation.

In fact here, the generalization algorithm can add further conditions. Since the generalization algorithm generalizes

**Case ¬ p[$y_0$,$z_0$]:**

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

$$\forall_{x,\text{it}[Y,Z]} (Y \approx Z \Rightarrow \text{cgg2}[x, Y] \approx \text{cgg2}[x, Z])$$

$$\forall_{y,\bar{y},z,\bar{z}} \langle y, \bar{y}, z, \bar{z}\rangle \approx \text{cgg2}[z, \langle y, \bar{y}, \bar{z}\rangle]$$

$$\forall_{y,\bar{y},z,\bar{z},\text{is–tuple}[Y]} \left( \bigwedge \left\{ \begin{array}{l} \text{is–sorted}[Y] \\ Y \approx \langle y, \bar{y}, \bar{z}\rangle \\ \text{is–sorted}[\langle y, \bar{y}\rangle] \end{array} \right\} \Rightarrow \text{is–sorted}[\text{cgg2}[z, Y]] \wedge \left\{ \begin{array}{l} \text{is–sorted}[\langle z, \bar{z}\rangle] \\ \neg\, p[y, z] \end{array} \right. \right).$$

## ■ 3.7. Summary of the Requirements for the Subalgorithms

We now collect all the requirements on the subalgorithms cee, ceg, cge, cgg1, cgg2:

$$\text{is–tuple[cee]}$$

$$\forall_{x,\bar{x}}\ \text{is–tuple[ceg[x, } \bar{x}\text{]]}$$

$$\forall_{x,\bar{x}}\ \text{is–tuple[cge[x, } \bar{x}\text{]]}$$

$$\forall_{x,\text{is–tuple[X]}} \left\{ \begin{array}{l} \text{is–tuple[cgg1[x, X]]} \\ \text{is–tuple[cgg2[x, X]]} \end{array} \right\}$$

$$\text{cee} = \langle\rangle$$

$$\forall_{z}\ (z \in \text{ceg[z]} \wedge (\text{dfo[z, ceg[z]]} = \langle\rangle))$$

$$\forall_{z}\ (\text{is–sorted[ceg[z]]})$$

$$\forall_{z0,z1,\bar{z}}\ (z0 \in \text{ceg[z0, z1, } \bar{z}\text{]} \wedge (\text{ceg[z1, } \bar{z}\text{]} \approx \text{dfo[z0, ceg[z0, z1, } \bar{z}\text{]]}))$$

$$\forall_{z0,z1,\bar{z}} \left( \left\{ \begin{array}{l} \text{is–sorted[ceg[z1, } \bar{z}\text{]]} \\ z0 \geq z1 \end{array} \right. \Rightarrow \text{is–sorted[ceg[z0, z1, } \bar{z}\text{]]} \right)$$

$$\forall_{y}\ (\text{is–sorted[cge[y]]})$$

$$\forall_{y0,y1,\bar{y}}\ (y0 \in \text{cge[y0, y1, } \bar{y}\text{]} \wedge (\text{ceg[y1, } \bar{y}\text{]} \approx \text{dfo[y0, ceg[y0, y1, } \bar{y}\text{]]}))$$

$$\forall_{y0,y1,\bar{y}} \left( \left\{ \begin{array}{l} \text{is–sorted[ceg[y1, } \bar{y}\text{]]} \\ y0 \geq y1 \end{array} \right. \Rightarrow \text{is–sorted[ceg[y0, y1, } \bar{y}\text{]]} \right)$$

$$\forall_{x,\text{is–tuple[Y,Z]}}\ (Y \approx Z \Rightarrow \text{cgg1[x, Y]} \approx \text{cgg1[x, Z]})$$

$$\forall_{y,\bar{y},z,\bar{z}}\ \langle y, \bar{y}, z, \bar{z}\rangle \approx \text{cgg1[y, } \langle \bar{y}, z, \bar{z}\rangle\text{]}$$

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

$$\forall_{x,\text{is–tuple[Y,Z]}}\ (Y \approx Z \Rightarrow \text{cgg2[x, Y]} \approx \text{cgg2[x, Z]})$$

$$\forall_{y,\bar{y},z,\bar{z}}\ \langle y, \bar{y}, z, \bar{z}\rangle \approx \text{cgg2[z, } \langle y, \bar{y}, \bar{z}\rangle\text{]}$$

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

Now it is clear that we have:

$$\text{cee} = \langle\rangle$$

and from

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

we get that in this 'ceg' takes a single element into the singleton tuple and from

$$\forall_{z} (is\text{–}sorted[ceg[z]])$$

we get that 'ceg' is the tuple constructor for single elements. However this is enough to select it as a candidate for our ceg function. And it is easy to check that this candidate verifies all the requirements. Similarly for 'cge':

$$\forall_{\overline{x}} (ceg[\overline{x}] = \langle \overline{x} \rangle)$$

$$\forall_{\overline{y}} (cge[\overline{y}] = \langle \overline{y} \rangle)$$

Now for 'cgg1', 'cgg2'. In the previous case study, we distinguished between empty tuples, singletons (together, the trivial tuples) and nontrivial tuples. Let's see what happens for singleton inputs.

$$c[\langle y\rangle, \langle z\rangle] = \left\{ \begin{matrix} cgg1[x, c[\langle\rangle, \langle y\rangle]] \Leftarrow p[x, y] \\ cgg2[y, c[\langle x\rangle, \langle\rangle]] \Leftarrow \neg\, p[x, y] \end{matrix} \right\} \leftrightarrow c[\langle x\rangle, \langle y\rangle] = \left\{ \begin{matrix} cgg1[x, \langle y\rangle] \Leftarrow p[x, y] \\ cgg2[y, \langle x\rangle] \Leftarrow \neg\, p[x, y] \end{matrix} \right\}$$

Now from the requirements of c, cgg1, cgg2, we can infer that the result is a tuple with two elements, which has to be sorted. From the definition of 'is–sorted' we then can infer that 'p' is '≥' (and it is necessarily the only candidate).

The only 'open' functions are now 'cgg1' and 'cgg2'. We can either look for functions that meet the requirements in the knowledge base, or try to synthesise them. It is easy to prove that the requirements are met by the concatenation function:

$$\forall_{x, is\text{–}tuple[Z]} (cgg1[x, Z] = x \frown Z)$$

$$\forall_{y, is\text{–}tuple[Z]} (cgg2[y, Z] = y \frown Z).$$

(If one embarks on synthesizing cgg1, cgg2 from the elementary tuple operations available in the knowledge base, it is again advisable to first simplify the above requirements to a few essential ones using simple symbolic computation proof techniques, without induction.)

## ■ 3.8. Synthesis of cee

### ■ Specification of the problem

$$is\text{–}tuple[cee]$$

$$cee = \langle\rangle$$

The specification is the definition, we are done

## ■ 3.9. Synthesis/discovery of ceg

### ■ Specification of the problem

$$\forall_{x,\bar{x}}\ \text{is–tuple}[\text{ceg}[x,\ \bar{x}]] \quad \text{"(ceg1)"}$$

$$\forall_{z}\ (z \in \text{ceg}[z] \wedge (\text{dfo}[z,\ \text{ceg}[z]] = \langle\rangle)) \quad \text{"(ceg2)"}$$

$$\forall_{z}\ (\text{is–sorted}[\text{ceg}[z]]) \quad \text{"(ceg3)"}$$

$$\forall_{z0,z1,\bar{z}}\ (z0 \in \text{ceg}[z0,\ z1,\ \bar{z}] \wedge (\text{ceg}[z1,\ \bar{z}] \approx \text{dfo}[z0,\ \text{ceg}[z0,\ z1,\ \bar{z}]])) \quad \text{"(ceg4)"}$$

$$\forall_{z0,z1,\bar{z}}\ \left(\left\{\begin{array}{l}\text{is–sorted}[\text{ceg}[z1,\ \bar{z}]] \\ z0 \geq z1\end{array}\right. \Rightarrow \text{is–sorted}[\text{ceg}[z0,\ z1,\ \bar{z}]]\right) \quad \text{"(ceg5)"}$$

### ■ Realization of *ceg* - Look into the knowledge base.

Look in the knowledge base. Study the function for the special input case (singleton), select a candidate and see if it fulfills the specifications.

In the knowledge base, we have

$$\forall_{x}\ \text{dfo}[x,\ \langle x\rangle] = \langle\rangle$$

from the definition of *dfo*. We also have

$$\forall_{x}\ x \in \langle x\rangle$$

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

All of these suggest the tuple identity function as the candidate for *ceg* (the identity already satisfies (ceg1)-(ceg3)). It is easy to show that indeed, it also satisfies (ceg4), (ceg5).

## ■ 3.10. Synthesis/discovery of cge

### ■ Specification of the problem

$$\forall_{y,\bar{y}}\ (\text{is–tuple}[\text{cge}[y,\ \bar{y}]])$$

$$\forall_{y}\ (\text{is–sorted}[\text{cge}[y]])$$

$$\forall_{y0,y1,\bar{y}}\ (y0 \in \text{cge}[y0,\ y1,\ \bar{y}] \wedge (\text{ceg}[y1,\ \bar{y}] \approx \text{dfo}[y0,\ \text{ceg}[y0,\ y1,\ \bar{y}]]))$$

$$\forall_{y0,y1,\bar{y}}\ \left(\left\{\begin{array}{l}\text{is–sorted}[\text{ceg}[y1,\ \bar{y}]] \\ y0 \geq y1\end{array}\right. \Rightarrow \text{is–sorted}[\text{ceg}[y0,\ y1,\ \bar{y}]]\right)$$

### ■ Realization of *cge*

Analogously as in the previous case, it is easy to show that the candidate function tuple identity satisfies the requirements for *cge*.

## ■ 3.11. Synthesis/discovery of cgg1

### ■ Specification of the problem

$$\forall_{x,\,is\text{–}tuple[X]} \text{is–tuple}[\text{cgg1}[x,\ X]] \quad \text{"(cgg1.1)"}$$

$$\forall_{x,is\text{–}tuple[Y,Z]} (Y \approx Z \Rightarrow \text{cgg1}[x, Y] \approx \text{cgg1}[x, Z]) \quad \text{"(cgg1.2)"}$$

$$\forall_{y,\bar{y},z,\bar{z}} \langle y, \bar{y}, z, \bar{z}\rangle \approx \text{cgg1}[y, \langle \bar{y}, z, \bar{z}\rangle] \quad \text{"(cgg1.3)"}$$

$$\forall_{y,\bar{y},z,\bar{z},is\text{–}tuple[Y]} \left( \bigwedge \left\{ \begin{array}{l} \text{is–sorted}[Y] \\ Y \approx \langle \bar{y}, z, \bar{z}\rangle \\ \text{is–sorted}[\langle y, \bar{y}\rangle] \end{array} \right\} \Rightarrow \text{is–sorted}[\text{cgg1}[y, Y]] \right) \quad \text{"(cgg1.4)"}$$

### ■ Realization of the algorithm-By looking at the Knowledge Base

Which are the functions that satisfy the type specification:

• binary functions: first argument and object, second tuple, and result a tuple.

**Remark:** before going on with the verification of the properties, let us look at the special case when the tuple argument is a singleton:

(cgg1.4) becomes:

$$\forall_{y,z,is\text{–}tuple[Y]} \left( \bigwedge \left\{ \begin{array}{l} \text{is–sorted}[Y] \\ Y \approx \langle z\rangle \\ \text{is–sorted}[\langle y\rangle] \end{array} \right\} \Rightarrow \text{is–sorted}[\text{cgg1}[y, Y]] \right)$$

from here, by the definition of ≈, we can deduce that Y=⟨z⟩, and since singletons are sorted, we get:

$$\forall_{y,\,z} (p[y,\ z] \Rightarrow \text{is–sorted}[\text{cgg1}[y,\ \langle z\rangle]]) \quad \text{"(cgg1.5)"}$$

Candidates:

dfo[x, X]

x ⌣ X

X ⌢ x (in fact this does not fit)

## ■ Candidate: dfo

It is easy to show that *dfo* satisfies (cgg1.1) and (cgg1.2) .

It is also easy to show that it does not satisfy (cgg1.3) - in fact this is always false for *dfo*.

Thus this candidate is rejected.

## ■ Candidate: prepend

(cgg1.1)-(cgg1.3) are satisfied by the prepend function.

Now for (cgg1.5):

We have:

$$\underset{y,z}{\forall} (p[y,\ z] \Rightarrow \text{is–sorted}[y \smile \langle z\rangle]) \leftrightarrow \underset{y,z}{\forall} (p[y,\ z] \Rightarrow \text{is–sorted}[\langle y,\ z\rangle])$$

From the definition of *is–sorted*, it follows that the relation *p* has to be ≥. This, of course, provided that append verifies (cgg1.4):

$$\underset{y,\overline{y},z,\overline{z},\text{is–tuple}[X]}{\forall} \left( \bigwedge \begin{cases} \text{is–sorted}[X] \\ X \approx \langle \overline{y},\ z,\ \overline{z}\rangle \\ \text{is–sorted}[\langle y,\ \overline{y}\rangle] \\ \text{is–sorted}[\langle z,\ \overline{z}\rangle] \\ y \geq z \end{cases} \Bigg\} \Rightarrow \text{is–sorted}[y \smile X] \right)$$

We proceed by case distinction on X:

**Proof.**

<u>Case X=⟨⟩:</u>
Assume for $\overline{y_0}$, $z_0$, $\overline{z_0}$:

$$\bigwedge \begin{cases} \text{is–sorted}[\langle\rangle] \\ \langle\rangle \approx \langle \overline{y_0},\ z_0,\ \overline{z_0}\rangle \\ \text{is–sorted}[\langle y_0,\ \overline{y_0}\rangle] \\ \text{is–sorted}[\langle z_0,\ \overline{z_0}\rangle] \\ y_0 \geq z_0 \end{cases} \Bigg\}$$

Prove

$$\text{is–sorted}[\langle y_0\rangle]$$

This is true since there is a contradiction in the assumptions.

<u>Case X singleton:</u>
Take $x_0$, $\overline{y_0}$, $z_0$, $\overline{z_0}$ arbitrary but fixed:

Assume:

$$\bigwedge \begin{Bmatrix} \text{is–sorted}[\langle x_0 \rangle] \\ \langle x_0 \rangle \approx \langle \overline{y},\ z_0,\ \overline{z_0} \rangle \\ \text{is–sorted}[\langle y_0,\ \overline{y_0} \rangle] \\ \text{is–sorted}[\langle z_0,\ \overline{z_0} \rangle] \\ y_0 \geq z_0 \end{Bmatrix}$$

and prove

$$\text{is–sorted}[\langle y_0,\ x_0 \rangle]$$

This is true! (definitions, etc etc).

Case X nontrivial:
Take $x_0$ , $x_1$ , $\overline{x_0}$ $\overline{y_0}$ , $z_0$ , $\overline{z_0}$ arbitrary but fixed.
Assume:

$$\bigwedge \begin{Bmatrix} \text{is–sorted}[\langle x_0,\ x_1,\ \overline{x_0} \rangle] \\ \langle x_0,\ x_1,\ \overline{x_0} \rangle \approx \langle \overline{y_0},\ z_0,\ \overline{z_0} \rangle \\ \text{is–sorted}[\langle y_0,\ \overline{y_0} \rangle] \\ \text{is–sorted}[\langle z_0,\ \overline{z_0} \rangle] \\ y_0 \geq z_0 \end{Bmatrix} \quad \text{"(A)"}$$

Show:

$$\text{is–sorted}[\langle y_0,\ x_0,\ x_1,\ \overline{x_0} \rangle]$$

By the definition of *is–sorted*, we have to show:

$$y_0 \geq x_0$$

$$\text{is–sorted}[\langle x_0,\ x_1,\ \overline{x_0} \rangle]$$

The last goal is the same with (A.1)

Now from (A.3)- (A.5), from the property

$$\forall_{x,\overline{x}} \left(\text{is–sorted}[\langle x,\ \overline{x} \rangle] \Rightarrow x \geq \max[\langle x,\ \overline{x} \rangle]\right)$$

and the properties of *max*, we have that

$$y_0 \geq \max[\langle y_0,\ \overline{y_0} \rangle]$$

$$y_0 \geq \max[\langle z_0,\ \overline{z_0} \rangle]$$

and from the properties

$$\forall_{a,\ \text{is–tuple}[X,\ Y]} \left(a \geq \max[X] \wedge a \geq \max[Y] \Rightarrow a \geq \max[X \asymp Y]\right)$$

$$\forall_{\text{is–tuple}[X,\ Y]} \left(X \approx Y \Rightarrow (\max[X] = \max[Y])\right)$$

and (A1)-(A2) we have:

$$y_0 \geq \max[\langle x_0,\ x_1,\ \overline{x_0} \rangle] \ (= x_0)$$

We are done! Prepend , $\smile$, satisfies the specification for *cgg1*.

### ■ Candidate: append

Apart from not fitting the type description, it can be shown easily that append does not satisfy the requirements.

## ■ 3.12. Synthesis/discovery of cgg2

### ■ Specification of the problem

$$\forall_{x,\,is\text{–}tuple[X]} \text{is–tuple}[\text{cgg2}[x,\ X]] \quad \text{(cgg2 .1)}$$

$$\forall_{x,is\text{–}tuple[Y,Z]} (Y \approx Z \Rightarrow \text{cgg2}[x, Y] \approx \text{cgg2}[x, Z]) \quad \text{(cgg2 .2)}$$

$$\forall_{y,\bar{y},z,\bar{z}} \langle y, \bar{y}, z, \bar{z}\rangle \approx \text{cgg2}[z, \langle y, \bar{y}, \bar{z}\rangle] \quad \text{(cgg2 .3)}$$

$$\forall_{y,\bar{y},z,\bar{z},is\text{–}tuple[Y]} \left( \bigwedge \left\{ \begin{array}{l} \text{is–sorted}[Y] \\ Y \approx \langle y, \bar{y}, \bar{z}\rangle \\ \text{is–sorted}[\langle y, \bar{y}\rangle] \ \} \Rightarrow \text{is–sorted}[\text{cgg2}[z, Y]] \\ \text{is–sorted}[\langle z, \bar{z}\rangle] \\ \neg\, p[y, z] \end{array} \right. \right) \quad \text{(cgg2 .4)}$$

**Remark:** before going on with the verification of the properties, let us again look at the special case when the tuple argument is a singleton:

(cgg2.4) becomes:

$$\forall_{y,z,is\text{–}tuple[Y]} \left( \bigwedge \left\{ \begin{array}{l} \text{is–sorted}[Y] \\ Y \approx \langle z\rangle \\ \text{is–sorted}[\langle y\rangle] \ \} \Rightarrow \text{is–sorted}[\text{cgg2}[z, Y]] \\ \text{is–sorted}[\langle z\rangle] \\ \neg\, p[y, z] \end{array} \right. \right)$$

from here, by the definition of $\approx$, we can deduce that $Y=\langle z\rangle$, and since singletons are sorted, we get:

$$\forall_{y,\,z} (\neg\, p[y,\ z] \Rightarrow \text{is–sorted}[\text{cgg1}[z,\ \langle y\rangle]]) \quad \text{(cgg2 .5)}$$

Like in the previous case, we have a few (the same candidates) in the knowledge base. For the case of *dfo* and *append* ($\frown$) it is easy to show that they do not satisfy the requirements.

Analogous to the previous case (*cgg1*), the *prepend* function can be shown to satisfy the requirements (with the same relation $\geq$ like in the previous case).

## ■ 3.13. Summary

Starting from the requirements for the combine function generated by the lazy thinking method for algorithm synthesis, we arrived at the following :

• the algorithm scheme for the combine function:

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

$$\forall_{y, \bar{y}} \; c[\langle\rangle, \langle y, \bar{y}\rangle] = ceg[y, \bar{y}]$$

$$\forall_{x, \bar{x}} \; c[\langle x, \bar{x}\rangle, \langle\rangle] = cge[x, \bar{x}]$$

$$\forall_{x, \bar{x}, y, \bar{y}} \; c[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle] = \left\{ \begin{array}{l} cgg1[x, c[\langle \bar{x}\rangle, \langle y, \bar{y}\rangle]] \Leftarrow p[x, y] \\ cgg2[y, c[\langle x, \bar{x}\rangle, \langle \bar{y}\rangle]] \Leftarrow \neg\, p[x, y] \end{array} \right\}$$

• auxiliary algorithms from the knowledge base:

$$cee = \langle\rangle$$

$$\forall_{x, \bar{x}} \; ceg[x, \bar{x}] = \langle x, \bar{x}\rangle$$

$$\forall_{y, \bar{y}} \; cge[y, \bar{y}] = \langle y, \bar{y}\rangle$$

$$\forall_{x,\, is\text{–}tuple[X]} \; cgg1[x, X] = x \smile X$$

$$\forall_{x,\, is\text{–}tuple[X]} \; cgg2[x, X] = x \smile X$$

p is the greater equal relation on elements of the tuples

Thus, we discovered the following algorithm for combining the tuples:

$$\text{Algorithm}\Big[\text{"combine (merge)"},\ any[x, \bar{x}, y, \bar{y}],$$

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

$$\forall_{y, \bar{y}} \; c[\langle\rangle, \langle y, \bar{y}\rangle] = \langle y, \bar{y}\rangle$$

$$\forall_{x, \bar{x}} \; c[\langle x, \bar{x}\rangle, \langle\rangle] = \langle x, \bar{x}\rangle$$

$$\forall_{x, \bar{x}, y, \bar{y}} \; c[\langle x, \bar{x}\rangle, \langle y, \bar{y}\rangle] = \left\{ \begin{array}{l} x \smile c[\langle \bar{x}\rangle, \langle y, \bar{y}\rangle] \Leftarrow x \geq y \\ y \smile c[\langle x, \bar{x}\rangle, \langle \bar{y}\rangle] \Leftarrow \neg\, x \geq y \end{array} \right\}$$

$$\Big]$$

# ■ 4. Synthesis of the Splitting Functions

## ■ 4.1. Specification of the problem

After the application of the lazy thinking method for algorithm synthesis for the sorting problem, the following specifications resulted for the splitting functions:

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

$$\forall_{\substack{is\text{–}tuple[X] \\ \neg is\text{–}trivial\text{–}tuple[X]}} \; (L[X] \asymp R[X]) \approx X$$

## ■ 4.2. Algorithm Scheme

Let us analyze the input to the functions: nontrivial tuples. Thus we will have a special case of:

$$\langle x,\ y\rangle$$

and a nonspecial case

$$\langle x,\ y,\ z,\ \overline{z}\rangle$$

**Remark.** There is the question of uncoupling the specifications. The point of uncoupling is to show that the problems are independent. In fact this is not the case here. The requirement that the concatenation of the splits gives back the argument tuple shows that they are not independent. In fact, splitting can be seen just like one function:

$$\text{Tuple} \to (\text{Tuple},\ \text{Tuple})$$

or given one of the splits, by the concatenation requirement we have the other one.

Now taking into the account the special case mentioned above, we could have the following scheme for the splits:

$$\forall_{x,y}\ (L[\langle x,\ y\rangle] = \text{spL1}[x, y])$$

$$\forall_{x,y,z,\overline{z}}\ (L[\langle x,\ y,\ z,\ \overline{z}\rangle] = \text{gL}[\langle x,\ y,\ z,\ \overline{z}\rangle])$$

$$\forall_{x,y}\ (R[\langle x,\ y\rangle] = \text{spR1}[x,\ y])$$

$$\forall_{x,y,z,\overline{z}}\ (R[\langle x,\ y,\ z,\ \overline{z}\rangle] = \text{gR}[\langle x,\ y,\ z,\ \overline{z}\rangle])$$

where gL and gR are some RHS's we do not discuss at this point (possible recursive).

### ■ Special Case ⟨x, y⟩

Let us postpone the discussion on the general form of the splits, and instead to analyze the problem for the special cases, namely prove that *spL1* and *spR1* are correct for the input, namely:

**Proof attempt begin.**

Prove, for arbitrary but fixed $x_0$, $y_0$:

$$\begin{cases} \text{is–tuple}[\text{spL1}[x_0,\ y_0]] \\ \text{spL1}[x_0,\ y_0] \prec \langle x_0,\ y_0\rangle \\ \text{is–tuple}[\text{spR1}[x_0,\ y_0]] \\ \text{spR1}[x_0,\ y_0] \prec \langle x_0,\ y_0\rangle \\ \text{spL1}[x_0,\ y_0] \asymp \text{spR2}[x_0,\ y_0] = \langle x_0,\ y_0\rangle \end{cases}$$

Starting with the last goal, since we are in a special case, we have the following possible decompositions:

$$\langle x_0,\ y_0\rangle$$

$$= \langle x_0,\ y_0\rangle \asymp \langle\rangle \quad (1)$$

$= \langle x_0 \rangle \asymp \langle y_0 \rangle$ (2)

$= \langle\rangle \asymp \langle x_0,\ y_0 \rangle$ (3)

$= \langle y_0 \rangle \asymp \langle x_0 \rangle$ (4)

And now we unify (1)-(4) with the last goal.

It is obvious that we cannot have (1) or (3), since this would make false the goal. With (1) and (4) we can prove the goal. We will only pursue (1), since (4) is similar.

The analysis done was exhaustive, so we can conclude:

$$\forall_{x,y} \bigwedge \begin{cases} L[\langle x,\ y\rangle] = \langle x\rangle \\ R[\langle x,\ y\rangle] = \langle y\rangle \end{cases}$$

**Remark.** We could choose:

• either of the (2) or (4) - and thus get 2 variants of splits - in fact the same, because once we got one, we got the other, automatically)

• or make our choice precise through a (unknown) relation we impose on x and y. (In this case, possibly this can lead to quicksort?! - will this become even more apparent with the rest of the exploration?).

## ■ Further Analysis: Algorithm Scheme variant 1: Second Special Case

Now for the *gL* and *gR* functions. Given the functional nature of the problem it is very likely that the splitting functions will be recursive, with the bottom of the recursion hitting one of the special cases. So we will probably have something like:

$L[\langle x,\ y,\ z,\ \bar{z}\rangle] = cL[L[\langle x,\ y\rangle],\ L[\langle z,\ \bar{z}\rangle]]$

$R[\langle x,\ y,\ z,\ \bar{z}\rangle] = cR[R[\langle x,\ y\rangle],\ R[\langle z,\ \bar{z}\rangle]]$

where *cL* and *cR* are unknown functions that combine the result of splitting the special situation with the rest of the splitting problem.

But from the above scheme we see that in case $\bar{z}$ is the empty sequence, the problem is not defined (we cannot split singletons). Thus we need to consider a second special case,

$L[\langle x,\ y,\ z\rangle] = spL2[\langle x,\ y,\ z\rangle]$

$R[\langle x,\ y,\ z\rangle] = spR2[\langle x,\ y,\ z\rangle]$

We now apply the lazy thinking method to synthesise *spL2* and *spR2*.

We attempt to prove the correctness theorem, i.e. that *spL2* and *spR2* satisfy the specifications for splits:

**Proof attempt begin.**

Prove, for arbitrary but fixed $x_0$, $y_0$, $z_0$:

$$\begin{cases} \text{is–tuple}[\text{spL2}[x_0,\ y_0,\ z_0]] \\ \text{spL2}[x_0,\ y_0,\ z_0] \prec \langle x_0,\ y_0,\ z_0\rangle \\ \text{is–tuple}[\text{spR2}[x_0,\ y_0,\ z_0]] \\ \text{spR2}[x_0,\ y_0,\ z_0] \prec \langle x_0,\ y_0,\ z_0\rangle \\ \text{spL2}[x_0,\ y_0,\ z_0] \asymp \text{spR2}[x_0,\ y_0,\ z_0] = \langle x_0,\ y_0,\ z_0\rangle \end{cases}$$

Like in the previous case, we start from the last goal. Since this is a special case, we can enumerate the possibilities:

$\langle x_0,\ y_0,\ z_0\rangle =$

$\langle x_0,\ y_0,\ z_0\rangle \asymp \langle\rangle$ (1)

$\langle x_0,\ y_0\rangle \asymp \langle z_0\rangle$ (2)

$\langle x_0\rangle \asymp \langle y_0,\ z_0\rangle$ (3)

$\langle\rangle \asymp \langle x_0,\ y_0,\ z_0\rangle$ (4)

... (all the possible permutations, but essentially variants of (1) – (4))

We can discard immediately (1) and (4), since they do not verify the requirements. Also, it is easy to show that (2) and (3) do.

We can now choose one of the decompositions, for instance

$$\forall_{x,y,z} \bigwedge \begin{cases} L[\langle x,\ y,\ z\rangle] = \langle x,\ z\rangle \\ R[\langle x,\ y,\ z\rangle] = \langle y\rangle \end{cases}$$

### ■ Summary: Algorithm Type for Splits, with 2 Special Cases

To summarize, we have the following algorithm scheme:

$$\forall_{x,y} \bigwedge \begin{cases} L[\langle x,\ y\rangle] = \langle x\rangle \\ R[\langle x,\ y\rangle] = \langle y\rangle \end{cases}$$

$$\forall_{x,y,z} \bigwedge \begin{cases} L[\langle x,\ y,\ z\rangle] = \langle x,\ z\rangle \\ R[\langle x,\ y,\ z\rangle] = \langle y\rangle \end{cases}$$

$$\forall_{x,y,z,t,\bar{z}} \bigwedge \begin{cases} L[\langle x,\ y,\ z,\ t,\ \bar{z}\rangle] = cL[\langle x\rangle,\ L[\langle z,\ t,\ \bar{z}\rangle]] \\ R[\langle x,\ y,\ z,\ t,\ \bar{z}\rangle] = cR[\langle y\rangle,\ R[\langle z,\ t,\ \bar{z}\rangle]] \end{cases}$$

$$\forall_{x,y,z,t,\bar{z}} \bigwedge \begin{cases} L[\langle x,\ y,\ z,\ t,\ \bar{z}\rangle] = cL[\langle x\rangle,\ L[\langle z,\ t,\ \bar{z}\rangle]] \\ R[\langle x,\ y,\ z,\ t,\ \bar{z}\rangle] = cR[\langle y\rangle,\ R[\langle z,\ t,\ \bar{z}\rangle]] \end{cases}$$

### ■ Algorithm Scheme Variant 2 (coming soon)

## ■ 4.3. Algorithm Synthesis by Lazy Thinking - using first algorithm scheme.

We have to prove the correctness theorem for the splits, namely that splits satisfy the requirements:

$$\forall_{\substack{\text{is–tuple[X]}\\ \neg\text{is–trivial–tuple[X]}}} \begin{cases} \text{L[X]} \prec \text{X} \\ \text{is–tuple[L[X]]} \\ \text{R[X]} < \text{X} \\ \text{is–tuple[R[X]]} \\ (\text{L[X]} \asymp \text{R[X]}) \approx \text{X} \end{cases}$$

Under the knowledge given by the algorithm type:

$$\forall_{x,y} \bigwedge \begin{cases} \text{L}[\langle x,\ y\rangle] = \langle x\rangle \\ \text{R}[\langle x,\ y\rangle] = \langle y\rangle \end{cases}$$

$$\forall_{x,y,z} \bigwedge \begin{cases} \text{L}[\langle x,\ y,\ z\rangle] = \langle x,\ z\rangle \\ \text{R}[\langle x,\ y,\ z\rangle] = \langle y\rangle \end{cases}$$

$$\forall_{x,y,z,t,\overline{z}} \bigwedge \begin{cases} \text{L}[\langle x,\ y,\ z,\ t,\ \overline{z}\rangle] = \text{cL}[\langle x\rangle,\ \text{L}[\langle z,\ t,\ \overline{z}\rangle]] \\ \text{R}[\langle x,\ y,\ z,\ t,\ \overline{z}\rangle] = \text{cR}[\langle y\rangle,\ \text{R}[\langle z,\ t,\ \overline{z}\rangle]] \end{cases}$$

and basic theory of tuples. In the algorithm type, cL and cR are unknown functions, with the following type specifications:

$$\forall_{x,\text{is–tuple[X]}} \bigwedge \begin{cases} \text{is–tuple[cL}[\langle x\rangle, \text{X]]} \\ \text{is–tuple[cR}[\langle x\rangle, \text{X]]} \end{cases}$$

The proof method used here is the following induction scheme (for nontrivial tuples):

In order to prove:

$$\forall_{\substack{\text{is–tuple[X]}\\ \text{is–trivial–tuple[X]}}} \text{P[X]}$$

1. Show, for arbitrary but fixed $x_0$, $y_0$, P[$\langle x_0, y_0\rangle$];

2. Show, for arbitrary but fixed $x_0$, $y_0$, $z_0$, P[$\langle x_0, y_0, z_0\rangle$];

3. For arbitrary but fixed $x_0$, $y_0$, $z_0$, $t_0$, $\overline{z_0}$,
   Assume P[$\langle z_0,\ t_0,\ z_0\rangle$];
   Show P[$\langle x_0, y_0, z_0, t_0, \overline{z_0}\rangle$].

## ■ First proof attempt

Already we have shown how this works for the special cases (1., 2.) and how we apply lazy thinking - in the discussion on the algorithm type for splits. Let us now move to the induction step.

Let $x_0$, $y_0$, $z_0$, $\overline{z_0}$ be arbitrary but fixed.

Assume

$$\bigwedge \begin{cases} \text{L}[\langle z_0,\ t_0,\ \overline{z_0}\rangle] \prec \langle z_0,\ t_0,\ \overline{z_0}\rangle \\ \text{is–tuple[L}[\langle z_0,\ t_0,\ \overline{z_0}\rangle]] \\ \text{R}[\langle z_0,\ t_0,\ \overline{z_0}\rangle] < \langle z_0,\ t_0,\ \overline{z_0}\rangle \\ \text{is–tuple[R}[\langle z_0,\ t_0,\ \overline{z_0}\rangle]] \\ (\text{L}[\langle z_0,\ t_0,\ \overline{z_0}\rangle] \asymp \text{R}[\langle z_0,\ t_0,\ \overline{z_0}\rangle]) \approx \langle z_0,\ t_0,\ \overline{z_0}\rangle \end{cases} \quad \text{(A)}$$

Show

$$\bigwedge \begin{cases} L[\langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle] \prec \langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle \\ \text{is–tuple}[L[\langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle]] \\ R[\langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle] \prec \langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle \\ \text{is–tuple}[R[\langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle]] \\ (L[\langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle] \asymp R[\langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle]) \approx \langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle \end{cases}$$

Which, by expansion of the definitions of *cL*, *cR*, becomes:

$$\bigwedge \begin{cases} cL[\langle x_0\rangle,\ L[\langle y_0,\ z_0, t_0,\ \overline{z_0}\rangle]] \prec \langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle \\ \text{is–tuple}[cL[\langle x_0\rangle,\ L[\langle y_0,\ z_0, t_0,\ \overline{z_0}\rangle]]] \\ R[\langle y_0\rangle, L[\langle z_0, t_0,\ \overline{z_0}\rangle]] \prec \langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle \\ \text{is–tuple}[R[\langle y_0\rangle, R[\langle z_0, t_0,\ \overline{z_0}\rangle]]] \\ (cL[\langle x_0\rangle,\ L[\langle z_0, t_0,\ \overline{z_0}\rangle]] \asymp cR[\langle x_0\rangle,\ R[\langle z_0, t_0,\ \overline{z_0}\rangle]]) \approx \langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle \end{cases} \qquad \text{(G)}$$

(G2) and (G4) follow immediately from the type specification

We are stuck proving (G2), (G4), (G5). By applying the Conjecture Generation Algorithm, with the generalization rules:

$x_0 \longrightarrow x$

$y_0 \longrightarrow y$

$L[\langle z_0,\ t_0,\ \overline{z_0}\rangle] \longrightarrow Y,\ \text{is–tuple}[Y]$

$R[\langle z_0,\ t_0,\ \overline{z_0}\rangle] \longrightarrow Z,\ \text{is–tuple}[Z]$

$\langle z_0,\ t_0,\ \overline{z_0}\rangle \longrightarrow X,\ \text{is–tuple}[X],\ \neg\ \text{is–trivial–tuple}[X]$

$\langle x_0,\ y_0,\ z_0, t_0,\ \overline{z_0}\rangle = \langle x_0,\ y_0\rangle \asymp \langle z_0, t_0,\ \overline{z_0}\rangle \longrightarrow_{\text{using previous generalizations}} \langle x,\ y\rangle \asymp X$

The following conjecture is generated:

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

## ■ 4.4. Synthesis/Discovery of cL, cR

### ■ Candidates

We are looking for functions

$$T \times T \to T$$

Candidate(s):

$$\asymp$$

### ■ Check: Concatenation Meets the Requirement

We will need the following properties:

$$\forall_{\text{is–tuple}[X, Y, Z, T]} \left( \left( \bigwedge \begin{cases} X \prec Y \\ Z \prec T \end{cases} \right) \Rightarrow (X \asymp Y) \prec (Y \asymp T) \right)$$

$$\forall_{\text{is–tuple}[X, Y, Z]} (X \asymp Y) \asymp Z \approx X \asymp (Y \asymp Z)$$

$$\forall_{\text{is–tuple}[X, Y]} (X \asymp Y) \approx (Y \asymp X)$$

Then it is easily possible to show that the concatenation meets the specifications:

Take $x_0$, $y_0$, $X_0$, $Y_0$, $Z_0$ arbitrary but fixed.

$$\bigwedge \begin{cases} \langle x_0 \rangle \prec \langle x_0,\ y_0 \rangle \\ Y \prec X \end{cases} \Rightarrow (\langle x_0 \rangle \asymp Y) \prec (\langle x, y \rangle \asymp X)$$

$$\bigwedge \begin{cases} \langle y_0 \rangle \prec \langle x_0,\ y_0 \rangle \\ Z \prec X \end{cases} \Rightarrow (\langle y_0 \rangle \asymp Z) \prec (\langle x,\ y \rangle \asymp X)$$

according to the conservation property of concatenation w.r.t the well founded ordering.

$$((\langle x_0 \rangle \asymp Y) \asymp (\langle y_0 \rangle \asymp Z)) \approx \langle x_0,\ y_0 \rangle \asymp ((\langle x_0,\ y_0 \rangle \asymp (Y \asymp Z)) \approx ((\langle x_0,\ y_0 \rangle \asymp X)$$

by rewriting using the associativity and commutativity of concatenation w.r.t. permutation of tuples.

## ■ 5. Conclusions

• Note that all the algorithm schemes used in the case studies have essentially the same structure (can be seen as instances of a divide–and–conquer scheme).

• Once selected a scheme, an analysis of the trivial cases is essential, i.e. we cannot tell in advance which is the trivial case.

• Proofs are rather long, sometimes the analysis of the trivial cases suggests particular (and peculiar) induction schemes. The problem of choosing the right induction scheme is not trivial, as is the availability of certain schemes in a system (*Theorema* in this case) used for the proofs needed for synthesis.

• For the retrieving aspect, type match candidates appeared to be a good choice. However, perhaps there is a need for more, as in more complex cases we could deal with a great number of candidates (or maybe not?). In particular it is (maybe a not very motivating) example what types would be useful for in the *Theorema* language.

## ■ References

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

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

[Basin et al. 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 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 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. Algorithm Retrieval: Concept Clarification and Case Study in *Theorema*. SFB Scientific Computing, Technical Report 2003-25, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, October 2003.

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

[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. 2000]
B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger. *Theorema*: 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 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, May 2003.

[Farmer 2002]
W. Farmer (ed.). The First North American Workshop on MKM (NA-MKM 2002) in Hamilton, Ontario, Canada, http://imps.mcmaster.ca/na-mkm-2002/.

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

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

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

[Merrit, Lau 1997]
S.M. Merritt, K.K. Lau. A Logical Inverted Taxonomy of Sorting Algorithms. In S. Kuru, M.U. Caglayan, and H.L. Akin, editors, Proceedings of the Twelfth International Symposium on Computer and Information Sciences, Bogazici University, pp. 576-583, 1997.

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

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

[Smith 1990]
D.R. Smith. KIDS: A semiautomatic program development system. IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16(9): 1024-1043, 1990.

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

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