# Preprocessed Lazy Thinking: Synthesis of Sorting Algorithms

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

In this paper we present an improvement of the lazy thinking method for program synthesis. We apply the lazy thinking method to a certain general class of problems (explicit problems) using a certain general algorithm scheme (divide–and–conquer) thus obtaining specifications for the subalgorithms of the scheme. This can then be used in particular case studies - like the synthesis of tuples discussed in this paper - without the need to repeat the induction proof.

## ■ 1. Introduction

This work was stimulated by some related results of N.Popov and T. Jebelean [Popov, Jebelean] in program verification. Using the technique of Scott induction (induction on the number of recursive calls), they have developed a tool to obtain sufficient verification results for a certain class of recursive programs, specified by input-output conditions. The general approach in deductive synthesis is to try a verification proof of the yet-unknown algorithm, which should lead to its instantiation. In this context Popov and Jebelean's approach is definitely interesting.

We tried to compare the use of this result for synthesis with the lazy thinking method. We thus applied lazy thinking to the problem and algorithm scheme and we realised that by doing so we can improve the practical use of lazy thinking. This report discusses these details.

First, a short account of Popov and Jebelean's approach is given, then we describe how we apply lazy thinking to the same problem and algorithm scheme, and we compare the two approaches. Finally we will show how the preprocessed lazy thinking can be applied practically, in the case study of synthesizing algorithms for sorting tuples.

## ■ 2. Verification of Functional Programs with Multiple Recursive Calls (General Verification Conditions - from [Popov, Jebelean 2004])

(Functional) Program with preconditions (input conditions) and postconditions (output conditions):

$$\{I\}\ F\ \{O\}$$

Partial correctness of the program $F$ with input conditions $I$ and output condition $O$:

$$\forall_n\ (F[n]\downarrow \wedge I[n] \Rightarrow O[n, F[n]])$$

Informally, if the program terminates (↓) and the argument satisfies the input condition, then the output condition is satisfied by the result.

Now, for a certain class of (functional) programs, namely recursive programs of the form:

$$F[n] = \begin{cases} q[n] & \Leftarrow \quad Q[n] \\ g[n,\ F[h_1[n]],\ \ldots,\ F[h_k[n]]] & \Leftarrow \quad \text{otherwise} \end{cases}$$

For proving partial correctness it sufices to show:

$$\forall_a\ (I[a] \wedge q[a]\downarrow \wedge Q[a] \Rightarrow O[a, q[a]])$$

$$\forall_a\ (I[a] \wedge \neg\, Q[a] \wedge h_1[a]\downarrow \wedge \ldots \wedge h_k[a]\downarrow \Rightarrow\ I[h_1[a]] \wedge \ldots \wedge I[h_k[a]]\ )$$

$$\forall_{a, b_1, \ldots, b_k} (I[a] \wedge \neg\, Q[a] \wedge h_1[a]\downarrow \wedge \ldots \wedge h_k[a]\downarrow \wedge$$
$$g[a, b_1, \ldots,\ b_k]\downarrow \wedge O[h_1[a], b_1] \wedge \ldots \wedge O[h_k[a], b_k] \wedge \ldots \Rightarrow O[a, g[a, b_1, \ldots, b_k]])$$

The above results are justified by a proof by the technique of Scott induction.

**Remarks:**

Note that the general recursive scheme presented above is the most general form of the 'divide–and–conquer' algorithm type (scheme). Of course, this has many applications and if we could use this in some way to synthesise 'divide-and-conquer' type algorithms, this would be rather useful.

In general, one of the most common approaches to program synthesis is to start a correctness proof for the intended algotrithm and exploit the failures to instantiate the algorithm. The method described by N. Popov would then allow to use this general method for any problem of the type described above and that can be solved using a 'divide-and-conquer' algorithm type. Compared to the lazy thinking synthesis method this seems to have the advantage that we can use the method without having to carry out any proof (safe for the initial Scott induction proof carried out to get these results). However, looking at the ingredients of this method - a general algorithm scheme that solves a general problem scheme - we could try to apply the lazy thinking method to these. In the following section we will detail this.

# ■ 3. Synthesis by Lazy Thinking of a Recursive Solution to an Abstract Problem on Inductive Domains

## ■ 3.1. The Abstract Problem

We are looking for an algorithm *F* to solve the following (explicit) problem:

$$\forall_{I[X]}\ O[X,\ F[X]]$$

where

I[X] - is the input condition, in our case describing an inductive domain;
O[X, F[X]] - the output condition.

Let us now consider the following general scheme scheme for the algorithm F:

$$F[X] = \begin{cases} s[X] & \Leftarrow \quad Q[X] \\ c[X,\ F[h_1[X]],\ \ldots,\ F[h_k[X]]] & \Leftarrow \quad \text{otherwise} \end{cases}$$

where

Q[X] is a predicate defining special objects in the chosen domain;

$h_1, \ldots, h_k$ are preprocessing functions;

c is a postprocessing function (that combines the result of the recursive calls) .

In order to be able to apply the lazy thinking method we need an induction rule attached to the scheme. This will be well–founded induction. Let thus $\prec$ be a well–founded ordering on the inductive domain of arguments. We attach the following specifications for the preprocessing functions to the algorithm scheme:

$$\forall_{\substack{I[X] \\ \neg Q[X]}} \bigwedge \begin{cases} h_1[X] \prec X \\ \ldots \\ h_k[X] \prec X \end{cases} .$$

The general scheme chosen here can be seen as a "divide–and–conquer" scheme, which encodes the following strategy to solving a given problem: if the input argument is simple, then solve the problem in a simple way, otherwise decompose the problem in "smaller" problems, solve them recursively and compose the result back. By adding the requirement that the subproblems are smaller, we also ensure the termination of the algorithm.

## ■ 3.2. Lazy Thinking: Failure Analysis and Conjecture Generation Algorithm - Reminder

In order for the lazy thinking method to be applicable (either in an automated way or by a human "pen and paper", "screen and keyboard" prover) the proving system must have certain features:

- proofs (proof objects) must be produced even in case of failure;
- the proof style must be natural.

For instance, Theorema is such a system.

In the case of a failing proof, the **failing proof situation** is given by a formula (implication) that has on the left hand side a conjunction of the temporary assumption made so far in the proof (tree), and on the right hand side the arbitrary goal.

Naturally, in the case there are no temporary assumption there is no implication. Also, according to the proof strategy of Theorema, all the ingredients of the failing proof situation will be (arbitrary but fixed) constants.

Once the failing proof situation is obtained, we employ the **conjecture generation algorithm**, which implements 2 conjecture generation strategies:

- in case in the failing proof situation we have something of the form $F[h_i[X_0]]$ (with the notation of the general algorithm scheme), we use the following generalization rule:

$F[h_i[X_0]] \longrightarrow X$

where X is a new variable;

- in any other cases, arbitrary but fixed constants $X_0$ are generalized to new variables X.

The conjecture generalization algorithm yields a list containing 2 lists of generalization rules, according to the 2 types of the generalizations presented above. The first one is applyed to the failing proof situation, then the second. The result is the conjecture that will allow the proof to go on.

**Remark.** Naturally, in the first case, the generalization has to be correct, so an effect of the application of the generalization rules of the first type we will get conditions on the functions $h_i$, namely that the result of their application to the input argument of the function F is closed with respect with the input condition for F. So in order to use the generalization rule

$$F[h_i[X_0]] \longrightarrow X$$

the following has to hold:

$$\forall_{I[X] \wedge \neg Q[X]} I[h_i[X]].$$

## ■ 3.3. Lazy Thinking on the Abstract Problem

We now attempt to prove the following (partial) correctness formula:

$$\forall_{I[X]} O[X, F[X]]$$

with knowledge on the above algorithm scheme and additional requirements.

### ■ Proof attempt: special case

We attempt to prove by well–founded induction with respect to $\prec$. Let $X_0$ be arbitrary but fixed such that $I[X_0]$.
We assume the induction hypothesis:

$$\forall_{\substack{I[Y] \\ Y \prec X_0}} O[Y, F[Y]]$$

and try to show:

$$O[X_0, F[X_0]]$$

Case: $Q[X_0]$.

By definition of F, we have to prove

$$O[X_0, s[X_0]].$$

The proof fails here. (The induction hypothesis is not applicable here).

### ■ Failing Proof Analysis + Conjecture Generation

We collect in one formula the temporary assumptions and the failing goal:

$$I[X_0] \wedge Q[X_0] \Rightarrow O[X_0, s[X_0]]$$

apply the conjecture generation algorithm with the following generalization rule:

$$X_0 \rightarrow \text{NewVariable}[X]$$

and we get

$$\forall_{\substack{I[X] \\ Q[X]}} O[X, s[X]]$$

which is a specification for the special function. By adding this to the knowledge base, we can get over the failure, and we continue with the proof.

## ■ Proof attempt continued: general case

Case: $\neg Q[X_0]$.

By definition of F, we have to prove:

$$O[X_0,\ c[X_0,\ F[h_1[X_0]],\ \ldots,\ F[h_k[X_0]]]].\quad (G)$$

By modus ponens, from the additional requirements on the preprocessing functions and the well–founded induction hypothesis we add to the knowledge base:

$$O[h_1[X_0],\ F[h_1[X_0]]]$$

$$\ldots$$

$$O[h_k[X_0],\ F[h_k[X_0]]]$$

The proof is stuck. We cannot make more progress.

## ■ Failing Proof Analysis + Conjecture Generation

The failing proof situation is given by the formula:

$$\left(\bigwedge\left\{\begin{array}{l} I[X_0] \\ \neg\, G[X_0] \\ O[h_1[X_0],\ F[h_1[X_0]]] \\ \ldots \\ O[h_k[X_0],\ F[h_k[X_0]]] \end{array}\right.\right) \Rightarrow O[X_0,\ c[X_0,\ F[h_1[X_0]],\ \ldots,\ F[h_k[X_0]]]]$$

Now the conjecture generation algorithm is applied. First we apply the following generalization rules:

$$F[h_1[X_0]] \to \text{NewVariable}[X_1]$$

$$\ldots$$

$$F[h_k[X_0]] \to \text{NewVariable}[X_k]$$

which can be applied provided the following holds:

$$\bigvee_{I[X] \wedge \neg Q[X]} \left(\bigwedge\left\{\begin{array}{l} I[h_1[X]] \\ \ldots \\ I[h_k[X]] \end{array}\right.\right).$$

After that we apply the generalization rule

$$X_0 \to \text{NewVariable}[X]$$

and we get the following conjecture:

$$\bigvee_{\substack{I[X, X_1, \ldots, X_k] \\ \neg Q[X]}} \left(\left(\bigwedge\left\{\begin{array}{l} O[h_1[X],\ X_1] \\ \ldots \\ O[h_k[X],\ X_k] \end{array}\right.\right) \Rightarrow O[X,\ c[X,\ X_1,\ \ldots,\ X_k]]\right)$$

provided the following also holds:

$$\forall_{I[X] \wedge \neg Q[X]} \left( \bigwedge \begin{Bmatrix} I[h_1[X]] \\ \dots \\ I[h_k[X]] \end{Bmatrix} \right).$$

Now with adding these conditions the proof goes through.

## ■ 3.4. Summary: Sufficient Proof Obligations and Use for Synthesis

Having carried the proof, we see that in order to prove the (partial) correctness of the algorithm F, it suffices to show:

$$\forall_{\substack{I[X] \\ Q[X]}} O[X,\ s[X]], \quad \text{(CC .1)}$$

$$\forall_{\substack{I[X] \\ \neg Q[X]}} \left( \bigwedge \begin{Bmatrix} I[h_1[X]] \\ \dots \\ I[h_k[X]] \end{Bmatrix} \right) \quad \text{(CC .2)}$$

and

$$\forall_{\substack{I[X, X_1, \dots, X_k] \\ \neg Q[X]}} \left( \left( \bigwedge \begin{Bmatrix} O[h_1[X],\ X_1] \\ \dots \\ O[h_k[X],\ X_k] \end{Bmatrix} \right) \Rightarrow O[X,\ c[X,\ X_1,\ \dots,\ X_k]] \right). \quad \text{(CC .3)}$$

with the additional requirement

$$\forall_{\substack{I[X] \\ \neg Q[X]}} \bigwedge \begin{Bmatrix} h_1[X] \prec X \\ \dots \\ h_k[X] \prec X \end{Bmatrix}. \quad \text{(CC s)}$$

In other words, we have a method to prove the (partial) correctness of algorithms F that are a solution for the problem (on an inductive domain)

$$\forall_{I[X]} O[X,\ F[X]]$$

and have the general form

$$F[X] = \begin{cases} s[X] & \Leftarrow \quad Q[X] \\ c[X,\ F[h_1[X]],\ \dots,\ F[h_k[X]]] & \Leftarrow \quad \text{otherwise} \end{cases}.$$

Moreover, considering the proof method we used, and the additional requirements on the preprocessing functions $h_1$, … , $h_k$ , we get the total correctness of the algorithm. Termination is ensured by the fact that the preprocessing functions make the input argument smaller parts (in a well–founded sense).

We can use this method for synthesis. The proof obligations correspond to the failing proof situations, and are, in fact, specifications for the auxiliary functions s, c, $h_1$, …, $h_k$. Of course, this only works if the conditions for applicability of the method are met, namely that the domain of the arguments is an inductive domain and the algorithm scheme is an instance of the scheme presented here.

### ■ 3.5. Comparison with Related Work

The results of N. Popov and T. Jebelean were one of the main reasons for this work. That paper is concerned with the verification or recursive programs that are defined by the exactly the same functional scheme as the one in this paper. The verification of such a program is done is two steps: first the partial correctness is proved, and then termination.

For the proof of partial correctness, the exact same sufficient conditions ((CC .1)-(CC .3)) are obtained (with additional termination requirements for the auxiliary functions on the left hand sides of the implications).
For the termination proof, the simplified version of the function is proved to be equal to a certain constant (in the same domain or projected in some other domain, where this proof might be simpler). This method is more general, because it can be applied on algorithms defined on non–inductive domains. The necessary conditions are obtained by a proof by Scott induction (on the number of recursive calls).

Potentially, this method for the verification of the algorithm can be applied also as a synthesis method, with the auxiliary algorithms in the scheme considered unknown. However, there are still problems concerned with the presence of the termination predicate on the unknown functions.

In contrast, the approach presented in this section, the application of the lazy thinking method to the general problem and the general recursive scheme, eliminates these difficulties, and ensures the termination of the synthesised algorithm. This approach will, however, require that algorithm is defined on an inductive domain and that we have a well–founded ordering on this domain. In the following we will show how to use this result, as a preprocessing step, for the case study of synthesis of sorting algorithms in the theory of tuples.

## ■ 4. Case Study: Sorting in the Theory of Tuples

### ■ 4.1. Specification of the problem

For solving the problem of sorting in the theory of tuples (synthesise the algorithm 's' which solves this) we start with the following specifications:

I : is–tuple[X]

O : is–sorted–version[X, s[X]]

and we employ the following scheme (from an algorithm schemes library):

$$\underset{\text{is–tuple}[X]}{\forall} s[X] = \begin{cases} sp[X] & \Longleftarrow \text{is–trivial–tuple}[X] \\ c[s[sp1[X]],\ s[sp2[X]]] & \Longleftarrow \text{otherwise} \end{cases}$$

where the marked subalgorithms are unknown, but should satisfy the following requirements:

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

$$\forall_{\substack{\text{is–tuple[X]} \\ \neg\text{is–trivial–tuple[X]}}} \bigwedge \begin{cases} \text{is–tuple[sp1[X]]} \\ \text{is–tuple[sp2[X]]} \end{cases}$$

$$\forall_{\substack{\text{is–tuple[X]} \\ \neg\text{is–trivial–tuple[X]}}} \bigwedge \begin{cases} X \succ \text{sp1[X]} \\ X \succ \text{sp2[X]} \end{cases}$$

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

## ■ 4.2. Application for the sorting problem

By the preprocessed lazy thinking, it suffices to prove:

$$\forall_X \ (\text{is–tuple[X]} \wedge \text{is–trivial–tuple[X]} \Rightarrow \text{isv[X, sp[X]]}) \quad \text{(G1)}$$

$$\forall_X \ (\text{is–tuple[X]} \wedge \neg \text{is–trivial–tuple[X]} \wedge \Rightarrow \text{is–tuple[sp1[X]]} \wedge \text{is–tuple[sp2[X]]}) \quad \text{(G2)}$$

$$\forall_{X,Y,Z} \ (\text{is–tuple[X]} \wedge \neg \text{is–trivial–tuple[X]} \wedge \text{isv[sp1[X], Y]} \wedge \text{isv[sp2[X], Z]} \Rightarrow \text{isv[X, c[Y, Z]]}) \quad \text{(G3)}$$

**Proof of (G1).**

(G1) is equivalent to:

$$\forall_{\substack{\text{is–tuple[X]} \\ \text{is–trivial–tuple[X]}}} \bigwedge \begin{cases} \text{is–tuple[sp[X]]} \\ X \approx \text{sp[X]} \\ \text{is–sorted[sp[X]]} \end{cases}$$

from here it follows easily the following requirement (definition) for 'sp'

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

**Proof of (G2).**

(G2) is equivalent to:

$$\forall_{\substack{\text{is–tuple[X]} \\ \neg\text{is–trivial–tuple[X]}}} \bigwedge \begin{cases} \text{is–tuple[sp1[X]]} \\ \text{is–tuple[sp2[X]]} \end{cases}$$

In fact this can be proved immediately if we consider that the type requirements are attached to the algorithm scheme. Otherwise it is a natural way to show how these type requirements appear.

**Proof of (G3).**

(G3) is equivalent to:

$$\forall_{\substack{\text{is–tuple[X, Y, Z]} \\ \neg\text{is–trivial–tuple[X]}}} \left( \bigwedge \begin{Bmatrix} \text{sp1[X]} \approx \text{Y} \\ \text{is–sorted[Y]} \\ \text{sp2[X]} \approx \text{Z} \\ \text{is–sorted[Z]} \end{Bmatrix} \Rightarrow \left( \bigwedge \begin{Bmatrix} \text{is–tuple[c[Y, Z]]} \\ \text{X} \approx \text{c[Y, Z]} \\ \text{is–sorted[c[Y, Z]]} \end{Bmatrix} \right) \right)$$

BINGO!!!!

This is the (coupled) specification (requirement) for the unknown functions!

# ■ 5. Coupled Specifications, Uncoupled Specifications

## ■ 5.1. Summary: coupled specifications for the splitting functions and for the composition

$$\forall_{\substack{\text{is–tuple[X]} \\ \neg\text{is–trivial–tuple[X]}}} \bigwedge \begin{Bmatrix} \text{is–tuple[sp1[X]]} \\ \text{is–tuple[sp2[X]]} \end{Bmatrix}$$

$$\forall_{\substack{\text{is–tuple[X]} \\ \neg\text{is–trivial–tuple[X]}}} \bigwedge \begin{Bmatrix} \text{is–tuple[sp1[X]]} \\ \text{is–tuple[sp2[X]]} \end{Bmatrix}$$

$$\forall_{\substack{\text{is–tuple[X, Y, Z]} \\ \neg\text{is–trivial–tuple[X]}}} \left( \bigwedge \begin{Bmatrix} \text{sp1[X]} \approx \text{Y} \\ \text{is–sorted[Y]} \\ \text{sp2[X]} \approx \text{Z} \\ \text{is–sorted[Z]} \end{Bmatrix} \Rightarrow \left( \bigwedge \begin{Bmatrix} \text{is–tuple[c[Y, Z]]} \\ \text{X} \approx \text{c[Y, Z]} \\ \text{is–sorted[c[Y, Z]]} \end{Bmatrix} \right) \right)$$

The decoupling of the specifications is discussed in [Craciun Buchberger 2004r].

## ■ 5.2. Decoupled specifications for the splitting functions (after some renaming)

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

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

## ■ 5.3. Decoupled specifications for the combining function

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

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

### ■ 5.4. Continue with the synthesis

Now try to "cast" the concrete situation into the general scheme  continue with the synthesis. We first do the splits, since it seem to be easier (only one argument for the function).

## ■ 6. Synthesis of Splits by Preprocessed Lazy Thinking

### ■ 6.1. Decoupled specifications for the splitting functions (after some renaming)

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

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

### ■ 6.2. Input/Output Conditions

$$\text{I}_{\text{L, R}} : \text{is–tuple[X]} \wedge \neg\, \text{is–trivial–tuple[X]}$$

$$\text{O}_{\text{L,R}} : \bigwedge \begin{cases} \text{is–tuple[L[X]]} \\ \text{L[X]} \prec \text{X} \\ \text{is–tuple[R[X]]} \\ \text{R[X]} \prec \text{X} \\ (\text{L[X]} \asymp \text{R[X]}) \approx \text{X} \end{cases}$$

### ■ 6.3. Recursion Scheme

In [Craciun, Buchberger 2004r] it is spelled out how one can get the following scheme after some investigation on "marginal cases":

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

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

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

Does this fit to the general scheme?

$$\text{F[n]} = \begin{cases} \text{q[n]} & \Leftarrow \text{Q[n]} \\ \text{g[n, F[h}_1\text{[n]], …, F[h}_\text{k}\text{[n]]]} & \Leftarrow \text{otherwise} \end{cases}$$

Well, it does. Take

$$h_1[\langle x,\ y,\ z,\ t,\ \overline{z}\rangle] = \langle x,\ y\rangle$$

$$h_2[\langle x,\ y,\ z,\ t,\ \overline{z}\rangle] = \langle z,\ t,\ \overline{z}\rangle$$

And we have 2 base cases.

Now we apply the preprocessed lazy thinking. To prove correctness, it is enough to prove:

$$\forall_{x,\,y} \begin{cases} \text{is–tuple}[L[\langle x,\ y\rangle]] \ (\leftrightarrow \text{is–tuple}[\langle x\rangle]) \\ L[\langle x,\ y\rangle] \prec \langle x,\ y\rangle \ (\leftrightarrow \langle x\rangle \prec \langle x,\ y\rangle) \\ \text{is–tuple}[R[\langle x,\ y\rangle]] \ (\leftrightarrow \text{is–tuple}[\langle y\rangle]) \\ R[\langle x,\ y\rangle] \prec \langle x,\ y\rangle \ (\leftrightarrow \langle y\rangle \prec \langle x,\ y\rangle) \\ (L[\langle x,\ y\rangle] \asymp R[\langle x,\ y\rangle]) \approx \langle x,\ y\rangle \ (\leftrightarrow\ (\langle x\rangle \asymp \langle y\rangle) \approx \langle x,\ y\rangle) \end{cases} \quad \text{(GSp1)}$$

$$\forall_{x,\,y,\,z} \begin{cases} \text{is–tuple}[L[\langle x,\ y,\ z\rangle]] \ (\leftrightarrow \text{is–tuple}[\langle x,\ z\rangle]) \\ L[\langle x,\ y,\ z\rangle] \prec \langle x,\ y,\ z\rangle \ (\leftrightarrow \langle x,\ z\rangle \prec \langle x,\ y,\ z\rangle) \\ \text{is–tuple}[R[\langle x,\ y,\ z\rangle]] \ (\leftrightarrow \text{is–tuple}[\langle y\rangle]) \\ R[\langle x,\ y,\ z\rangle] \prec \langle x,\ y,\ z\rangle \ (\leftrightarrow \langle y\rangle \prec \langle x,\ y,\ z\rangle) \\ (L[\langle x,\ y,\ z\rangle] \asymp R[\langle x,\ y,\ z\rangle]) \approx \langle x,\ y,\ z\rangle \ (\leftrightarrow\ (\langle x,\ z\rangle \asymp \langle y\rangle) \approx \langle x,\ y,\ z\rangle) \end{cases} \quad \text{(GSp1)}$$

$$\forall_{x,\,y,\,z,\,t,\,\overline{z}} \left( \bigwedge \begin{cases} \text{is–tuple}[\langle x,\ y\rangle] \\ \neg\ \text{is–trivial–tuple}[\langle x,\ y\rangle] \\ \text{is–tuple}[\langle z,\ t,\ \overline{z}\rangle] \\ \neg\ \text{is–trivial–tuple}[\langle z,\ t,\ \overline{z}\rangle] \end{cases} \right) \quad \text{(GSp3)}$$

$$\forall_{x,\,y,\,z,\,t,\,\overline{z},\,\text{is–tuple}[X,\,Y,\,Z,\,T]} \left( \bigwedge \begin{cases} \text{is–tuple}[X] \\ X \prec \langle x,\ y\rangle \\ \text{is–tuple}[Y] \\ Y \prec \langle z,\ t,\ \overline{z}\rangle \\ \text{is–tuple}[Z] \\ Z \prec \langle x,\ y\rangle \\ \text{is–tuple}[T] \\ T \prec \langle z,\ t,\ \overline{z}\rangle \\ (X \asymp Z) \approx \langle x,\ y\rangle \\ (Y \asymp T) \approx \langle z,\ t\,,\ \overline{z}\rangle \end{cases} \right) \Rightarrow \quad \text{(GSp4)}$$

$$\left( \bigwedge \begin{cases} \text{is–tuple}[cL[L[\langle x,\ y\rangle],\ L[\langle z,\ t,\ \overline{z}\rangle]]] \\ cL[L[\langle x,\ y\rangle],\ L[\langle z,\ t,\ \overline{z}\rangle]] \prec \langle x,\ y,\ z,\ t,\ \overline{z}\rangle \\ \text{is–tuple}[cR[R[\langle x,\ y\rangle],\ R[\langle z,\ t,\ \overline{z}\rangle]]] \\ cR[R[\langle x,\ y\rangle],\ R[\langle z,\ t,\ \overline{z}\rangle]] \prec \langle x,\ y,\ z,\ t,\ \overline{z}\rangle \\ (cL[L[\langle x,\ y\rangle],\ L[\langle z,\ t,\ \overline{z}\rangle]] \asymp cR[R[\langle x,\ y\rangle],\ R[\langle z,\ t,\ \overline{z}\rangle]]) \approx \langle x,\ y,\ z,\ t,\ \overline{z}\rangle \end{cases} \right)$$

Taking into account the "marginal cases", this goal can be further simplified:

$$\forall_{x,\,y,\,z,t,\overline{z},\,\text{is–tuple}[Y,\,T]} \left( \bigwedge \begin{Bmatrix} \text{is–tuple}[Y] \\ Y \prec \langle z,\ t,\ \overline{z} \rangle \\ \text{is–tuple}[T] \\ T \prec \langle z,\ t,\ \overline{z} \rangle \\ (Y \asymp T) \approx \langle z,\ t\,,\ \overline{z} \rangle \end{Bmatrix} \right) \Rightarrow \qquad \text{(GSp5)}$$

$$\left( \bigwedge \begin{Bmatrix} \text{is–tuple}[\text{cL}[\langle x \rangle,\ \text{L}[\langle z,\ t,\ \overline{z} \rangle]]] \\ \text{cL}[\langle x \rangle,\ \text{L}[\langle z,\ t,\ \overline{z} \rangle]] \prec \langle x,\ y,\ z,\ t,\ \overline{z} \rangle \\ \text{is–tuple}[\text{cR}[\langle y \rangle,\ \text{R}[\langle z,\ t,\ \overline{z} \rangle]]] \\ \text{cR}[\langle y \rangle,\ \text{R}[\langle z,\ t,\ \overline{z} \rangle]] \prec \langle x,\ y,\ z,\ t,\ \overline{z} \rangle \\ (\text{cL}[\langle x \rangle,\ \text{L}[\langle z,\ t,\ \overline{z} \rangle]] \asymp \text{cR}[\langle x \rangle,\ \text{R}[\langle z,\ t,\ \overline{z} \rangle]]) \approx \langle x,\ y,\ z,\ t,\ \overline{z} \rangle \end{Bmatrix} \right)$$

For the realization of cL and cR, see [Craciun Tma Report 1].

# ■ 7. Synthesis of Compose

## ■ 7. 1. Specifications for the combining function

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

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

## ■ 7.2. Input/Output Conditions

$$\text{I}_\text{M} : \text{is–tuple}[Y,\ Z] \wedge \text{is–sorted}[Y] \wedge \text{is–sorted}[Z]$$

$$\text{O}_\text{M} : \bigwedge \begin{Bmatrix} \text{M}[Y,\ Z] \approx (Y \asymp Z) \\ \text{is–sorted}[\text{M}[Y,\ Z]] \end{Bmatrix}$$

## ■ 7.3. Algorithm Scheme ([BuchbergerCraciun-2003 MKM])

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

$$\forall_{y,\,\overline{y}}\ \text{c}[\langle\rangle,\ \langle y,\ \overline{y} \rangle] = \text{ceg}[y,\ \overline{y}]$$

$$\forall_{x,\,\overline{x}}\ \text{c}[\langle x,\ \overline{x} \rangle,\ \langle\rangle] = \text{cge}[x,\ \overline{x}]$$

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

Does this fit to the general scheme?

For once, we have 2 parameters, which apparently does not fit to our general algorithm scheme.

There are 2 possible solutions:

1. Force casting this scheme into the general scheme introduced above.

2. Investigate the preprocessed lazy thinking for different schemes.

## ■ First Choice: Casting the algorithm scheme for the combining function into the general verification conditions

$$\forall_{x,y,\bar{x},\bar{y}} \left( \begin{array}{c} c[\langle\rangle,\ \langle\rangle] = cee \\ c[\langle\rangle,\ \langle y,\ \bar{y}\rangle] = ceg[y,\ \bar{y}] \\ c[\langle x,\ \bar{x}\rangle,\ \langle\rangle] = cge[x,\ \bar{x}] \\ c[\langle x,\ \bar{x}\rangle,\ \langle y,\ \bar{y}\rangle] = cgg[c[h_1[\langle x,\ \bar{x}\rangle,\ \langle y,\ \bar{y}\rangle]],\ c[h_2[\langle x,\ \bar{x}\rangle,\ \langle y,\ \bar{y}\rangle]]] \end{array} \right)$$

And now "cheating"* we can say that:

$$h_1[\langle x,\ \bar{x}\rangle,\ \langle y,\ \bar{y}\rangle] = \begin{cases} seq[\langle x\rangle,\ \langle\rangle] & \Leftarrow\ x \geq y \\ seq[\langle\rangle,\ \langle y\rangle] & \Leftarrow\ \text{otherwise} \end{cases}$$

$$h_2[\langle x,\ \bar{x}\rangle,\ \langle y,\ \bar{y}\rangle] = \begin{cases} seq[\langle \bar{x}\rangle,\ \langle y,\ \bar{y}\rangle] & \Leftarrow\ x \geq y \\ seq[\langle x,\ \bar{x}\rangle,\ \langle \bar{y}\rangle] & \Leftarrow\ \text{otherwise} \end{cases}$$

*call it educated guess, rather. It is easy to see that once the base cases have been synthesised , the scheme works. For details how to get the base cases, see [Craciun-MergeSynthesis]. But of course, only if the same happens with this case study.

## ■ Argument for the correctness of this casting

To be able to apply the general lazy thinking scheme, we must provide an inductive domain, and a well–founded (noetherian) ordering (in fact the noetherian ordering suffices, since it implies that the domain is inductive).

Our domain is made up of sequences of 2 tuples. Let us define the following ordering on this domain:

$$\forall_{X,Y,Z,W} (seq[X,\ Y] \prec_2 seq[Z,\ W] \Leftrightarrow (X \prec Z \wedge\ W \not\prec Y) \vee (Z \not\prec X \wedge Y \prec W))$$

By the fact that $\prec$ is Noetherian, $\prec_2$ is also Noetherian. The splitting functions defined above obviously make their output smaller in the sense of $\prec_2$ in report with their input.

## ■ Verification conditions for the base cases:

$$\bigwedge \begin{cases} cee \approx \langle\rangle \\ is\text{–}sorted[cee] \end{cases} \quad \text{(CBase1)}$$

$$\forall_{y,\bar{y}} (is\text{–}sorted[\langle y,\ \bar{y}\rangle]) \Rightarrow \left( \bigwedge \begin{cases} ceg[y,\ \bar{y}] \approx \langle y,\ \bar{y}\rangle \\ is\text{–}sorted[ceg[y,\ \bar{y}]] \end{cases} \right) \quad \text{(CBase2)}$$

$$\forall_{x,\bar{x}} (is\text{–}sorted[\langle x,\ \bar{x}\rangle]) \Rightarrow \left( \bigwedge \begin{cases} cge[x,\ \bar{x}] \approx \langle x,\ \bar{x}\rangle \\ is\text{–}sorted[cge[x,\ \bar{x}]] \end{cases} \right) \quad \text{(CBase3)}$$

## ■ Now compare with the specifications from [BuchbergerCraciun 2003 MKM]:

$$\forall_z (z \in ceg[z] \wedge (dfo[z,\ ceg[z]] = \langle\rangle))$$

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

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

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

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

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

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

Not the same, but are they equivalent?! The difference is that in the MKM paper, we used more knowledge from the background, before abandoning the proof attempt. However, applying the same knowledge to (CBase1), (CBase2), (CBase3) would lead to the same results.

This leads to the following definitions for the basic cases:

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

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

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

## ■ Now for the second condition

Here the input condition is more complicated. What is required is that the result of the application of the preprocessing functions be suitable input for the main algorithm. By choosing the output of $h_1$, $h_2$ to be a sequence of 2 tuples (in each case), then this condition is met.

## ■ Now for the third condition

For the first preprocessing function, the output condition is easy to verify. We will leave it out. We distinguish 2 cases:

$$\forall_{\substack{x,\bar{x},y,\bar{y},\, \text{is–tuple}[Y] \\ x \geq y}} \left( \bigwedge \begin{cases} \text{is–sorted}[\langle \bar{x} \rangle] \\ \text{is–sorted}[\langle y, \bar{y} \rangle] \end{cases} \Rightarrow \bigwedge \begin{cases} Y \approx \langle \bar{x}, y, \bar{y} \rangle \\ \text{is–sorted}[Y] \end{cases} \right) \Rightarrow$$

$$\left( \left( \bigwedge \begin{cases} \text{is–sorted}[\langle x, \bar{x} \rangle] \\ \text{is–sorted}[\langle y, \bar{y} \rangle] \end{cases} \right) \Rightarrow \bigwedge \begin{cases} \text{cgg}[\langle x \rangle, Y] \approx \langle x, \bar{x}, y, \bar{y} \rangle \\ \text{is–sorted}[\text{cgg}[\langle x \rangle, Y]] \end{cases} \right)$$

Now, using the fact that:

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

this is equivalent to:

$$\forall_{x,\bar{x},y,\bar{y},\, \text{is–tuple}[Y]} \left( \left( \bigwedge \begin{cases} x \geq y \\ \text{is–sorted}[\langle x, \bar{x} \rangle] \\ \text{is–sorted}[\langle y, \bar{y} \rangle] \\ \text{is–sorted}[Y] \\ Y \approx \langle \bar{x}, y, \bar{y} \rangle \end{cases} \right) \Rightarrow \left( \bigwedge \begin{cases} \text{cgg}[\langle x \rangle, Y] \approx \langle x, \bar{x}, y, \bar{y} \rangle \\ \text{is–sorted}[\text{cgg}[\langle x \rangle, Y]] \end{cases} \right) \right)$$

Analogously,

$$\forall_{x,\bar{x},y,\bar{y},\ \text{is–tuple}[Y]} \left( \left( \bigwedge \begin{Bmatrix} x < y \\ \text{is–sorted}[\langle x,\ \bar{x}\rangle] \\ \text{is–sorted}[\langle y,\ \bar{y}\rangle] \\ \text{is–sorted}[Y] \\ Y \approx \langle x,\ \bar{x},\ \bar{y}\rangle \end{Bmatrix} \Rightarrow \left( \bigwedge \begin{Bmatrix} \text{cgg}[\langle y\rangle,\ Y] \approx \langle x,\ \bar{x},\ y,\ \bar{y}\rangle \\ \text{is–sorted}[\text{cgg}[\langle y\rangle,\ Y]] \end{Bmatrix} \right) \right)$$

Again, this is the same result as in the case of the application of the lazy thinking method to the concrete problem and algorithm scheme.

## ■ 8. Conclusions

We have shown that by applying the method of lazy thinking to a general problem scheme and a general algorithm scheme we obtain a variant of the method that for practical purposes has the following advantages:

- cheaper:
  - → save the proof;
- no induction rule problems:
  - → no need to generate it automatically - it is implicit;
- no, or little, conjecture generation needed.

## ■ References

[Buchberger, Craciun 2003]
B. Buchberger, A. Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. in: Fairouz Kamareddine (ed.), Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, Nov. 25, 2003, Electronic Notes on Theoretical Computer Science, volume dedicated to the MKM 03 Symposium, Elsevier, ISBN 044451290X.

[Craciun, Buchberger 2004 a]
A. Craciun, B. Buchberger. Algorithm Synthesis Case Studies: Sorting of Tuples by Lazy Thinking. *Theorema* Technical Report, 04-01, 2004.

[Popov, Jebelean 2004]
N. Popov, T. Jebelean. Verification of Simple Recursive Programs: Sufficient Conditions. RISC Technical Report 04-06, RISC Linz, Austria, 2004