Analele Universităţii de Vest, Timişoara
Seria Matematică – Informatică
XLVIII, 3, (2010), 47– 58

A Case Study in Proof Based Synthesis of Sorting Algorithms

Isabela Drămnesc and Tudor Jebelean

Abstract. We present a case study in proof based constructive synthesis of sorting algorithms. Using a knowledge base containing the necessary properties of tuples, we start from the specification of the problem (input and output conditions) and we construct an inductive proof of the fact that for each input there exists a sorted tuple. During the proof our problem reduces into simpler and simpler problems, we apply the same method ("cascading") for the new problems and finally the problem is so simple that one can find the necessary functions in the knowledge base. The algorithm can be then extracted immediately from the proof.
We focus here on the synthesis of the merge-sort algorithm. This case study is paralleled with the exploration of the appropriate theory of tuples. The purpose of the case study is multifold: to construct the appropriate knowledge base necessary for this type of proofs, to find the natural deduction inference rules and the necessary strategies for their application, and finally to implement the corresponding prover in the frame of the *Theorema* system.
The novel specific feature of our approach is applying this method like in a "cascade" and the use (as much as possible) of first order predicate logic because this increases the feasibility of proving.

AMS Subject Classification (2000). 68Q60
Keywords. sorting algorithms synthesis merge sort

1 Introduction

The concern of algorithm synthesis is to develop methods and tools for mechanizing and automatizing (parts of) the process of finding an algorithm that satisfies a given specification. Although constructive logic already gives comprehensive methods for extracting algorithms from proofs, it is still a challenge to actually find such proofs for concrete problems. For an overview of several methods for synthesizing recursive programs in computational logic see [2].

In the present paper, by proving we synthesize the Merge-Sort algorithm. We show how one can reduce the original problem successively into simpler problems and by applying the same method like in a "cascade" finally the problem is so simple that the desired algorithm (function) already exists in the knowledge base. The process of proving is paralleled with the process of exploring the appropriate theory.

Related Work

Bruno Buchberger introduces some strategies for systematic theory exploration in a bottom-up and in a top-down way, see [3]. In fact many of the definitions and assumptions from tuple theory which we use in this paper are developed in that research work, see also [9].

Bruno Buchberger introduces a method for algorithm synthesis, called "lazy thinking", see [4] and also a model for theory exploration using schemes, see [5]. The "lazy thinking" method is implemented in the *Theorema* system [6], see [7]. In this approach, a "program schemata" (the structure of the solving term) is given to the synthesis procedure. In contrast, in our approach this schemata is also discovered by the procedure, only the inductive principle of the domain is given. Another approach for automatizing the synthesis of logic programs was introduced in [12].

Case studies on the synthesis of the Merge-Sort algorithm have been presented in [13], [14]. Our approach it better allows the automatization of the synthesis process. A case study on the synthesis of the Insertion-Sort algorithm for tuples using this method is presented in [10], [11].

In contrast with the other existent case studies on sorting algorithms from literature, in particular sorting of tuples, in our approach we use a different method for synthesizing the algorithms and they use as much as possible first order logic. This method was first introduced in 2008, for details see [8]. A case study of transforming recursive procedures into tail recursive using this method of synthesis can be found in [1].

2 Description of the method

In this section we give a general description of our approach.
We start from the specification of the problem (input and output condition). Given the *Problem Specification:*

Input: $I_F[X]$
Output: $O_F[X, Y]$

we want to find the definition of F such that $\underset{X:I_F}{\forall} O_F[X, F[X]]$. Note that in our formalism we use square brackets as in $f[x]$ for function application and for predicate application, instead of the usual round parantheses as in $f(x)$. Also, the quantified formula above is a notation for: $(\forall X)(I_F[X] \Longrightarrow O_F[X, F[X]])$.
We prove $\underset{X:I_F}{\forall}\underset{Y}{\exists} O_F[X, Y]$, either directly or using an induction principle (the choice of the proof style and of the induction principle is not automatic).
Direct proof. We take X_0 arbitrary but fixed (a new constant), we assume $I_F[X_0]$ and by proof we find a witness $T[X_0]$ (a term depending on X_0) such that $O_F[X_0, T[X_0]]$. In this case, the algorithm is $F[X] = T[X]$.
Inductive proof. Let us consider for illustration the head-tail inductive definition of tuples. In our notation, $\langle\rangle$ represents the empty tuple, $a \smile \langle\rangle$ is a tuple having a single element a, and $a \smile X$ is the tuple having head a and tail X. Complementary to this inductive definition, we consider that the functions Head and Tail are in the knowledge base. Moreover, in order for this induction scheme to be appropriate for our algorithm, the following properties must hold: $I_F[\langle\rangle]$, and $\underset{a}{\forall}\underset{X}{\forall} I_F[a \smile X] \Longrightarrow I_F[X]$. (That is: the input condition must hold for the base case and for the inductive decomposition of the argument.)
Base case: We prove $\underset{Y}{\exists} O_F[\langle\rangle, Y]$. If the proof succeeds to find a ground term C such that $O_F[\langle\rangle, C]$, then we know that $F[\langle\rangle] = C$.
Induction step: For new constants a and X_0 (satisfying I_F), we assume $\underset{Y}{\exists} O_F[X_0, Y]$ and we prove $\underset{Y}{\exists} O_F[a \smile X_0, Y]$. We transform the existential assumption by introducing a new constant Y_0 into: $O_F[X_0, Y_0]$. If the proof succeeds to find a witness $T[a, X_0, Y_0]$ (i.e., a term depending on a, X_0, and Y_0) such that $O_F[a \smile X_0, T[a, X_0, Y_0]]$, then we know that $F[a \smile X] = T[a, X, F[X]]$.
Algorithm extraction: Finally the algorithm is expressed as:

$$F[X] = \begin{cases} C, & \text{if } X = \langle\rangle \\ T[Head[X], Tail[X], F[Tail[X]]], & \text{if } X \neq \langle\rangle \end{cases}$$

Cascading. Assume that in the induction step above the prover cannot

find a witness. Then we can create the following new problem: "Given a, X, Y such that $I_F[X]$ and $O[X, Y]$, find a Z such that $O_F[a \smile X, Z]$." That is, if we cannot find an appropriate term, we try to synthesize (by a possibly different induction principle) a new function $G[a, X, Y]$ which does the job. Syntactically this new problem looks more complicated, however it is logically simpler, because the arguments fulfill more conditions, while the output requirement is the same as before. As we demonstrate below, this cascading principle can be repeated, and the problem will be reduced into simpler and simpler problems until the necessary functions are already in the knowledge base.
During the proof we discover propositions that we need for the proof in order to succeed. We prove these propositions separately and after that introduce them into the knowledge base and use them in the proof. We will not go into details in this paper for the process of inventing propositions. We just remark that by adding propositions to the knowledge base we explore the theory. Thus the process of proving is paralleled to the process of exploring (building) the theory, and the case study presented here increases our experience in developing a reasonable theory of tuples for further practical applications.

3 Synthesis of the Merge-Sort Algorithm

In this section we describe in detail the proof based synthesis of the Merge-Sort algorithm.
We start with the *Problem Specification*:

$$I_{\text{MergeSort}}[X] : \text{True}$$
$$O_{\text{MergeSort}}[X, Y] : \left\{ \begin{array}{c} X \approx Y \\ \text{IsSorted}[Y] \end{array} \right.$$

By the notation "$\approx$" we understand that X has the same elements with Y and by $IsSorted[Y]$ we understand that Y is sorted.
In the *knowledge base* we have the definitions: "$\approx$", "IsSorted", "$\lhd$" (an element occurs in a tuple) and the definition "concatenation" (of two tuples):

Definition. *"$\approx$":*

$$\underset{a,X,Y}{\forall} \left(\begin{array}{c} \langle\rangle \approx \langle\rangle \\ ((a \smile X) \approx Y) \Longleftrightarrow ((a \lhd Y) \wedge X \approx dfo[a, Y]) \end{array} \right)$$

Here "dfo" is the function that deletes the first occurrence of an element in a tuple and is:

Definition. *"dfo":*

$$\underset{a,b,X}{\forall}\left(\begin{array}{c} dfo[a,\langle\rangle]=\langle\rangle \\ dfo[a,a\frown X]=X \\ (a\neq b)\Longrightarrow(dfo[a,b\frown X]=b\frown dfo[a,X]) \end{array}\right)$$

Definition. *"IsSorted":*

$$\underset{a,b,X}{\forall}\left(\begin{array}{c} IsSorted[\langle\rangle] \\ IsSorted[a\frown\langle\rangle] \\ IsSorted[a\frown(b\frown X)]\Longleftrightarrow((a\leq b)\wedge \\ IsSorted[b\frown X]) \end{array}\right)$$

Definition. *"$\lhd$":*

$$\underset{a,b,X}{\forall}\left(\begin{array}{c} a\not\lhd\langle\rangle \\ a\lhd a\frown X \\ (a\neq b)\Longrightarrow((a\lhd b\frown X)\Longleftrightarrow(a\lhd X)) \end{array}\right)$$

Definition. *"concatenation":*

$$\underset{a,b,X}{\forall}\left(\begin{array}{c} \langle\rangle\asymp X=X \\ (a\frown X)\asymp Y=a\frown(X\asymp Y) \end{array}\right)$$

The problem specification and the properties above are essentially the same as in [3] – only that we do not use sequence variables.
In order to synthesize the sorting algorithm we try to prove:

Proposition. *"Problem of Sorting":*

$$\underset{XY}{\forall\exists}\left(\begin{array}{c} X\approx Y \\ IsSorted[Y] \end{array}\right)$$

Proof. It is sufficient to prove the proposition "Problem of Sorting" because we do not take into account the type of the objects. We consider that all the objects are tuples and the fact that we do not check the type of the objects does not influence our proof.
For proving we use the induction principle:

$$\Big(P[\langle\rangle]\bigwedge\underset{a}{\forall}(P[a\frown\langle\rangle])\bigwedge\underset{A,B}{\forall}((P[A]\bigwedge P[B])\Longrightarrow P[A\asymp B])\Big)$$
$$\Longrightarrow\underset{A,B}{\forall}P[A\asymp B]$$

This induction principle corresponds to the "divide–and–conquer" algorithm design scheme. The induction is in second order logic, but by instantiation this becomes first order. We assume that this comes together with two functions "*OddIP* " and "*EvenIP* " which split a tuple into two disjoint tuples. Of course there can be many such functions, but the choice is not important for the synthesis process. (Of course it may be important for the efficiency in certain environments.)

Definition. *"Elements on the Odd positions from a tuple"*

$$\underset{a,b,X}{\forall}\left(\begin{array}{c}(OddIP[\langle\rangle]=\langle\rangle)\\ OddIP[a\frown\langle\rangle]=a\frown\langle\rangle\\ OddIP[a\frown(b\frown X)]=a\frown OddIP[X]\end{array}\right)$$

Definition. *"Elements on the Even positions from a tuple"*

$$\underset{a,b,X}{\forall}\left(\begin{array}{c}EvenIP[\langle\rangle]=\langle\rangle\\ EvenIP[a\frown\langle\rangle]=\langle\rangle\\ EvenIP[a\frown(b\frown X)]=b\frown EvenIP[X]\end{array}\right)$$

The first two branches of the proof correspond to the cases: *Case 1:* X=$\langle\rangle$ and *Case 2:* $X = a \frown \langle\rangle$. These proofs are relatively simple (for details please see [11]) and they use inference steps like:

- In order to prove an existential goal on variable v, consider a meta–variable $v^?$ (a name for a term which we do not know) and continue the proof until a suitable value (term) for the metavariable is found.
- In order to prove the conjunction of two formulae containing the same meta–variable $P[v^?]$ and $Q[v^?]$, first try to find the witness term T for $P[v^?]$ and then prove $Q[T]$. (Of course some backtracking mechanism is necessary here.)
- Finding the term for a metavariable by matching or by unification.

Case 3: $X = A \asymp B$
Prove:

$$\left(\underset{Y_1}{\exists}\left(\begin{array}{c}A\approx Y_1\\ \text{IsSorted}\,[Y_1]\end{array}\right)\wedge\underset{Y_2}{\exists}\left(\begin{array}{c}B\approx Y_2\\ \text{IsSorted}\,[Y_2]\end{array}\right)\right)\Longrightarrow$$
$$\underset{Y}{\exists}\left(\begin{array}{c}(A\asymp B)\approx Y\\ \text{IsSorted}[Y]\end{array}\right)$$

We skolemize the two premises using new constants Y_1, Y_2 and in the goal we use the metavariable $Y^?$. The function (say M) which performs the merging $Y^? = M[Y_1, Y_2]$ may or may not be in our current knowledge base.
Case when we know the function M: When we have in the knowledge base the definition of the function that combines two sorted tuples, the prover will find the witness by matching the goal against the definition. We omit the presentation of this simpler case – see [11].
Case when we do not know the function M:
This is the situation when "cascading" occurs.
We need to synthesize the function that combines two sorted tuples into a sorted one.

The new *Problem Specification* is:

$$I_{\text{Merge}}\,[Y_1, Y_2] : \text{IsSorted}[Y_1],\ \text{IsSorted}[Y_2]$$
$$O_{\text{Merge}}\,[Y_1, Y_2, Y] : \left\{ \begin{array}{c} (Y_1 \asymp Y_2) \approx Y \\ \text{IsSorted}[Y] \end{array} \right.$$

Note how the original problem (of sorting a tuple) is reduced to a simpler one (merging two sorted tuples) because there are more assumptions about the input but the same requirement on the output.
We generate the conjecture:

Proposition. *"reduced problem MergeSort-1"*

$$\underset{Y_1,Y_2}{\forall}\left(\left(\begin{array}{c} \mathit{IsSorted}\,[Y_1] \\ \mathit{IsSorted}\,[Y_2] \end{array}\right) \Longrightarrow \underset{Y}{\exists}\left(\begin{array}{c} (Y_1 \asymp Y_2) \approx Y \\ \mathit{IsSorted}[Y] \end{array}\right)\right)$$

We try to prove this proposition using the head–tail induction principle for two variables:

$$\begin{aligned} \Big(P[\langle\rangle, \langle\rangle] \bigwedge \underset{a,X}{\forall}(P[X, \langle\rangle] \Longrightarrow P[a \smile X, \langle\rangle]) \bigwedge \\ \underset{b,Y}{\forall}(P[\langle\rangle, Y] \Longrightarrow P[\langle\rangle, b \smile Y]) \bigwedge \\ \underset{a,b,X,Y}{\forall}(P[X, Y] \Longrightarrow P[a \smile X, b \smile Y])\Big) \\ \Longrightarrow \underset{X,Y}{\forall} P[X, Y] \end{aligned}$$

So, we have 4 cases:

1) $Y_1 : \langle\rangle$ and $Y_2 : \langle\rangle$
2) $Y_1 : a \smile Y_1$ and $Y_2 : \langle\rangle$
3) $Y_1 : \langle\rangle$ and $Y_2 : b \smile Y_2$
4) $Y_1 : a \smile Y_1$ and $Y_2 : b \smile Y_2$

Case 1) $Y_1 : \langle\rangle$ and $Y_2 : \langle\rangle$

Prove: $\underset{Y}{\exists}\left(\begin{array}{c} (\langle\rangle \asymp \langle\rangle) \approx Y \\ \text{IsSorted}[Y] \end{array}\right)$ $(G4.1)$

Find $Y^?$ such that $\left\{ \begin{array}{ll} (\langle\rangle \asymp \langle\rangle) \approx Y^? & \text{(G4.11)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4.12)} \end{array} \right.$

We rewrite (G4.11) by the definition "concatenation" and we must prove:

$$\left\{ \begin{array}{ll} \langle\rangle \approx Y^? & \text{(G4.13)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4.12)} \end{array} \right.$$

By matching $\langle\rangle \approx \langle\rangle$ (from the definition "$\approx$") with $\langle\rangle \approx Y^?$ we obtain witness $Y^? = \langle\rangle$.
Check into (G4.12): IsSorted[$\langle\rangle$]. This is true by the definition "IsSorted".
Thus, we obtain $M[\langle\rangle, \langle\rangle] = \langle\rangle$ $(*I*)$.

Case 2) $Y_1 : a \smile Y_1$ and $Y_2 : \langle\rangle$

Assume: IsSorted$[a \smile Y_1]$ (H4.2)
Prove: $\underset{Y}{\exists}\begin{pmatrix} ((a \smile Y_1) \asymp \langle\rangle) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G4.2)
Find $Y^?$ such that:

$$\begin{cases} ((a \smile Y_1) \asymp \langle\rangle) \approx Y^? & \text{(G4.21)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4.22)} \end{cases}$$

We rewrite (G4.21) by the definition "concatenation" and we have to prove:

$$\begin{cases} (a \smile Y_1) \approx Y^? & \text{(G4.23)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4.22)} \end{cases}$$

By proposition "reflexivity in $\approx$" in (G4.23) we obtain the witness $Y^? = a \smile Y_1$.
The knowledge base must contain the proposition:

Proposition. *"reflexivity in* $\approx$ *"* $\underset{X}{\forall}(X \approx X)$

Check for (G4.22): IsSorted[a$\smile Y_1$]. This is true by (H4.2).
So, we obtain $M[a \smile Y_1, \langle\rangle] = a \smile Y_1$ $(*II*)$.

Case 3) $Y_1 : \langle\rangle$ and $Y_2 : b \smile Y_2$
Assume IsSorted$[b \smile Y_2]$ (H4.3)
Prove $\underset{Y}{\exists}\begin{pmatrix} ((\langle\rangle) \asymp b \smile Y_2) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G4.3)
Find $Y^?$ such that:

$$\begin{cases} ((\langle\rangle) \asymp b \smile Y_2) \approx Y^? & \text{(G4.31)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4.32)} \end{cases}$$

We rewrite (G4.31) by the definition "concatenation" and we have to prove:

$$\begin{cases} (b \smile Y_2) \approx Y^? & \text{(G4.33)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4.32)} \end{cases}$$

By proposition "reflexivity in $\approx$" in (G4.33) we obtain witness $Y^? = b \smile Y_2$.
Check for (G4.32): IsSorted[b$\smile Y_2$]. This is true by (H4.3).
So, we obtain $M[\langle\rangle, b \smile Y_2] = b \smile Y_2$ $(*III*)$.

Case 4) $Y_1 : a \smile Y_1$ and $Y_2 : b \smile Y_2$
Assume: IsSorted$[a \smile Y_1]$ (H4.41) and IsSorted$[b \smile Y_2]$ (H4.42)
Prove: $\underset{Y}{\exists}\begin{pmatrix} ((a \smile Y_1) \asymp (b \smile Y_2)) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G5)
We consider that $(a \smile Y_1) \asymp Y_2$ and $Y_1 \asymp (b \smile Y_2)$ are smaller than $(a \smile Y_1) \asymp (b \smile Y_2)$ and for proving (G5) we use the induction principle:

$$\underset{a,b,Y_1,Y_2}{\forall}\Big(P[(a \smile Y_1) \asymp Y_2] \bigwedge P[Y_1 \asymp (b \smile Y_2)]\Big) \Longrightarrow$$

$$\underset{a,b,Y_1,Y_2}{\forall} P[(a \frown Y_1) \asymp (b \frown Y_2)]$$

We reduce our problem into proving the proposition:

Proposition. *"reduced problem Merge 2.1"*

$$\left(\underset{Z_1}{\exists}\begin{pmatrix}(Y_1 \asymp (b \frown Y_2)) \approx Z_1\\ IsSorted[Z_1]\end{pmatrix} \wedge \underset{Z_2}{\exists}\begin{pmatrix}((a \frown Y_1) \asymp Y_2) \approx Z_2\\ IsSorted[Z_2]\end{pmatrix}\right) \Longrightarrow$$

$$\underset{Z}{\exists}\begin{pmatrix}((a \frown Y_1) \asymp (b \frown Y_2)) \approx Z\\ IsSorted[Z]\end{pmatrix}$$

This is the second "cascading" step in our synthesis process.

By skolemization of the premises we obtain:

$$\begin{cases}(Y_1 \asymp (b \frown Y_2)) \approx Z_1 & \text{(H5.1)}\\ \text{IsSorted}\,[Z_1] & \text{(H5.2)}\end{cases} \wedge \begin{cases}((a \frown Y_1) \asymp Y_2) \approx Z_2 & \text{(H6.1)}\\ \text{IsSorted}\,[Z_2] & \text{(H6.2)}\end{cases}$$

The goal becomes: find witness $Z^?$ such that

$$\begin{cases}((a \frown Y_1) \asymp (b \frown Y_2)) \approx Z^? & \text{(G5)}\\ \text{IsSorted}\left[Z^?\right] & \text{(G6)}\end{cases}$$

Alternative 1: Using the definition "concatenation" in (G5) we obtain $a \frown (Y_1 \asymp (b \frown Y_2)) \approx Z^?$ $(G5.1)$.

By $(Y_1 \asymp (b \frown Y_2)) \approx Z_1 (H5.1)$ we obtain $Z^? \approx a \frown Z_1$. By matching with the proposition "reflexivity in $\approx$" we obtain the witness: $Z^? = a \frown Z_1$.

Check for (G6): IsSorted $[a \frown Z_1]$. Check for $IsSorted[Z_1]$ which we know by (H5.2) and for $a \leq Z_1$ (denotes that a is smaller than all the elements of Z_1).

By $(Y_1 \asymp (b \frown Y_2)) \approx Z_1$ (H5.1) using the Proposition "*the smallest element1* " and the Proposition "*the smallest element2* " we have to check for $a \leq Y_1$ which is true by $IsSorted[a \frown Y_1](H4.41)$ and for $a \leq b \frown Y_2$. So, by the proposition "*the smallest element2* " we have to check for $a \leq b$ (this becomes the unknown assumption) and for $a \leq b \frown Y_2$. We know that $IsSorted[b \frown Y_2](H4.42)$ so, we obtain: $a \leq b \Longrightarrow Z^? = a \frown Z_1$ and $M\,[a \frown Y_1, b \frown Y_2] = a \frown Z_1$ $(*IV*)$.

The knowledge base must contain:

Proposition. *"the smallest element1"*

$$\underset{a,X,Y}{\forall}((a \leq X \wedge X \approx Y) \Longrightarrow a \leq Y)$$

Proposition. *"the smallest element2"*

$$\underset{a,X,Y}{\forall}((a \leq X \wedge a \leq Y) \Longrightarrow a \leq X \asymp Y)$$

Note that when we match a simple goal (like e.g. $a \leq b$) which cannot be prove or disprove we use it as an assumption in the algorithm and we generate another branch in which we assume that the opposite $\neg(a \leq b)$ is true.
Alternative 2: We match the proposition "reflexivity in $\approx$" with (G5) and obtain the witness: $Z^? = (a \frown Y_1) \asymp (b \frown Y_2)$. Using the proposition "*the same elements in concatenation*" we obtain $Z^? = b \frown ((a \frown Y_1) \asymp Y_2)$. The knowledge base must contain the proposition:

Proposition. *"the same elements in concatenation"*

$$\underset{a,X,Y}{\forall}(X \asymp (a \frown Y) \approx a \frown (X \asymp Y))$$

By (H6.1) we obtain the witness: $Z^? = b \frown Z_2$.
Check for (G6): IsSorted $[b \frown Z_2]$
By (H6.2), similar with the case before, we use the proposition "*order*" and assume that b<a is true and then we obtain that IsSorted $[b \frown Z_2]$ is true, that was our goal. The theory must contain the proposition:

Proposition. *"order"*

$$\underset{a,b}{\forall}\left(\ (\neg(a \leq b)) \Longrightarrow (b < a)\ \right)$$

So, we obtain: $b < a \Longrightarrow Z^? = b \frown Z_2$ and
$M\,[a \frown Y_1, b \frown Y_2] = b \frown Z_2$ $(*V*)$.
From $(*I*)$, $(*II*)$, $(*III*)$, $(*IV*)$ and $(*V*)$, by the transformation rules $(Z_1 \longrightarrow M\,[Y_1, b \frown Y_2],\ Z_2 \longrightarrow M\,[a \frown Y_1, Y_2])$ we obtain the well-known merging algorithm:

$$\underset{a,b,X,Y}{\forall}\left(\begin{array}{c} M[\langle\rangle, \langle\rangle] = \langle\rangle \\ M[a \frown X, \langle\rangle] = a \frown X \\ M[\langle\rangle, b \frown Y] = b \frown Y \\ M[a \frown X, b \frown Y] = \\ \left\{\begin{array}{c} a \frown M[X, b \frown Y] \Longleftarrow a \leq b \\ b \frown M[a \frown X, Y] \Longleftarrow \text{otherwise} \end{array}\right\} \end{array}\right)$$

From *Case1*, *Case2* and *Case3* we extract the merge-sort algorithm:

$$\underset{a,X}{\forall}\left(\begin{array}{c} \mathcal{S}[\langle\rangle] = \langle\rangle \\ \mathcal{S}[a \frown \langle\rangle] = a \frown \langle\rangle \\ \mathcal{S}[X] = M[\mathcal{S}[OddIP[X]], \mathcal{S}[EvenIP[X]]] \end{array}\right)$$

□

Note how by successive transformations of the problem we are able to reduce it to find witnesses which contain only elementary predicates and functions on tuples and on elements.

4 Conclusion and Further Work

The algorithm synthesis method presented here and the underlying proof methods are subject to implementation in the *Theorema* system. The case study shows that this method is effective and relatively efficient because by reducing the problem into simpler and simpler problems by applying the same method like in a "cascade" finally we obtain the desired algorithm directly from the knowledge base – the efficiency depends of course on the strength of the proof methods.
We developed similar case studies on other tuple operations and other sorting methods (like e. g. insertion sort), and we plan to extend this work by investigating further algorithms in a similar way (like e. g. quick sort).
The investigation of all these cases can constitute the basis of realizing a comprehensive theory of tuples, as well as a prover which is able to synthesize most of the algorithms, as well as to prove most of the conjectures. The work can then proceed in the direction of designing strategies for finding appropriate induction principles in the course of the synthesis process.

References

[1] **P. Audebaud and L. Chiarabini**, *New Development in Extracting Tail Recursive Programs from Proofs*, Pre-Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'09), Coimbra, Portugal, September 9-11, 2009, 2009.

[2] **D. Basin, Y. Deville, P. Flener, A. Hamfelt, J. F. Nillson, and M. Modelling**, *Synthesis of programs in computational logic*, Program Development in Computational Logic, Springer, 2004, 30-65.

[3] **B. Buchberger**, Theory Exploration with Theorema, *Analele Universitatii Din Timisoara, Ser. Matematica-Informatica*, **XXXVIII**, (2000), 9-32.

[4] **B. Buchberger**, Algorithm Invention and Verification by Lazy Thinking, *Analele Universitatii din Timisoara, Seria Matematica - Informatica*, **XLI**, (2003), 41-70.

[5] **B. Buchberger**, *Algorithm Supported Mathematical Theory Exploration: A Personal View and Strategy*, Proceedings of AISC 2004, Ed. by B. Buchberger Campbell, **3249**, Springer LNAI, 236-250.

[6] **B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger**, Theorema: Towards Computer-Aided Mathematical Theory Exploration, *Journal of Applied Logic*, **4** (4), (2006), 470-504.

[7] **B. Buchberger and A. Craciun**, Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema, *Electr. Notes Theor. Comput. Sci.*, **93**, (2004), 24-59.

[8] **L. Chiarabini**, *Extraction of Efficient Programs from Proofs: The Case of Structural Induction over Natural Numbers*, Local Proceedings of the Fourth Conference on Computability in Europe: Logic and Theory of Algorithms (CiE'08), Ed. by A. Beckmann and C. Dimitracopoulos and B. Löwe, Athens, Greece, June 15-20, 2008, 2008, 64-76.

[9] **I. Dramnesc, T. Jebelean, and A. Craciun**, *Case Studies in Systematic Exploration of Tuple Theory*, Proc. of RISC Report Series, University of Linz, Austria, Vol. 10-09, 2010.

[10] **I. Dramnesc, T. Jebelean, and A. Craciun**, *A Case Study in Systematic Exploration of Tuple Theory*, Proceedings of Symbolic Computation in Software Science (Hagenberg, Austria), Ed. by Tudor Jebelean and Mohamed Mosbah and Nikolaj Popov, Proc. of RISC Report Series, University of Linz, Austria, Vol. 10-10, 2010, 82-95.

[11] **I. Dramnesc and T. Jebelean**, *Proof Based Synthesis of Sorting Algorithms*, RISC Report Series, 10-17, Proc. of University of Linz, Austria, Vol. 10-17, 2010.

[12] **I. Kraan, D. Basin, and A. Bundy**, *Middle-Out Reasoning for Logic Program Synthesis*, Proc. 10th Intern. Conference on Logic Programing (ICLP '93) (Budapest, Hungary), MIT Press, Cambridge, MA, 1993, 441-455.

[13] **D. R. Smith**, *A Problem Reduction Approach to Program Synthesis*, IJCAI, 1983, 32-36.

[14] **D. R. Smith**, Top-down Synthesis of Divide-and-Conquer Algorithms, *Artificial Intelligence*, **27** (1), (1985), 43-96.

Isabela Drămnesc

Department of Computer Science
West University of Timişoara
V. Pârvan nr. 4, Timişoara,
Romania

E-mail: `idramnesc@info.uvt.ro`

Tudor Jebelean

Research Institute for Symbolic Computation
Johannes Kepler University, Linz
Austria

E-mail: `Tudor.Jebelean@jku.at`

Received: 30.06.2010

Accepted: 25.10.2010