Deductive Synthesis of Bubble–Sort Using Multisets

Isabela Drămnesc
Department of Computer Science
West University
Timişoara, Romania
Email: isabela.dramnesc@e-uvt.ro

Tudor Jebelean
Research Institute for Symbolic Computation,
Johannes Kepler University,
Linz, Austria
Email: Tudor.Jebelean@jku.at

***Abstract*—We demonstrate the possibility of automated synthesis of the *Bubble–Sort* algorithm as a rewrite program, a functional program, and an iterative program, starting from the specification. First a rewrite set of clauses for the algorithm *Max–Sort* is generated from the automatic proof of the synthesis conjecture, representing the main algorithm as well as the necessary auxiliary functions. This is then transformed into a tail recursive *Bubble–Sort* and by logical analysis the possibility of adding a flag for avoiding unnecessary recursions is identified. This new, more efficient algorithm is then transformed into a functional program, and finally into an imperative program. The practical experiments are performed using the *Theorema* system.**

***Index Terms*—automated reasoning, algorithm synthesis, lists, multisets, Theorema**

I. Introduction

Deductive synthesis[1] consists in producing a running algorithm or program on the basis of the specification of the intended functionality of it. The specification consists in an *input condition* $I[X]$ which characterizes the admissible inputs and an *output condition* $O[X, Y]$ which shows the relation between the inputs and the outputs. From this a *synthesis conjecture* is produced as the statement $\underset{X}{\forall}(I[X] \implies \underset{Y}{\exists} O[X, Y])$. From the proof of this conjecture (if certain constructive requirements are met) one extracts the algorithm. (Various approaches to the proof construction will generate various algorithms.) Typically an algorithm requires some auxiliary functions: by certain methods the synthesis conjectures for these can be generated during the main proof, and the auxiliary functions will be synthesized additionally (sometimes they require other auxiliary functions – thus one has a "cascading" of such synthesis proofs – see [2]).

In this paper we demonstrate the synthesis of *Bubble–Sort*, in the *Theorema* system [19] by the use of *multisets* and of certain proof techniques introduced in [9]. Moreover we describe the systematic transformation of the synthesized algorithm into a tail recursive one, then a systematic logical analysis in order to detect how to use a flag in order to avoid unnecessary recursions (the algorithm will stop when the sublist currently addressed is already sorted). This more efficient algorithm is transformed into a functional program and then into an imperative program. For the efficiency of the latter, we assume that the running environment provides an efficient implementation (constant time) of appending an element to a list and of concatenating two lists[2].

A. Related work and originality

The problem of algorithms synthesis, especially sorting algorithms is well-studied in the literature. [6] derives versions of six sorting algorithms (including *Bubble-Sort* called *Exchange-Sort*) by applying transformation rules. [1] extends [6] by adding another symmetry, see also [12]. [11] uses specific transformation techniques, which complement the techniques in [6], and derives several sorting algorithms (including *Bubble-Sort*). [15] uses a top–down approach to synthesize a large family of sorting algorithms (including *Bubble-Sort*). [15] designs also a tree which represents a classification of the sorting algorithms. [17] introduces deductive tableau techniques for algorithm synthesis. These techniques are applied in [18] to manually synthesize several sorting algorithms. [14] applies deductive tableau techniques [17], uses some heuristics and rippling [3] for automatically synthesize several functions in Lisp. Significant work also has been done in the proof–based algorithm synthesis on lists [8] and on binary trees [10].

In [13] the authors give a brief description of 8 iterative sorting algorithms (including *Bubble-Sort*, *Selection-Sort*, and *Heap-Sort*) and they propose a hardware optimization to be applied in order to speed–up the process of sorting. They also compare the performance of the 8 sorting algorithms with respect to resource usage and time execution. Most of researchers find the performance (space and time) of recursive algorithms as being lower compared to tail recursive and iterative algorithms.

The first study [5] on transforming recursive functions, including recursion removal [4], into more efficient ones uses some transformation rules and strategies. Their implementation in [7] is based on a set of schemas, whereas in [16] the authors use a different method which is based on incrementalization and is not schema–based.

The main novelty of our approach consists in using (a) *multisets* for expressing the fact that two lists have the same elements, in contrast to other approaches which use the predicate *perm* (a list is a permutation of another), as well

[1] An early description of this approach, including references to previous works, is [17].

[2] This can be easily realized, even if the representation of lists is done in *Lisp* style using pointers, by keeping additionally a pointer to the last element of the list

as (b) *patterns* for the arguments of functions, in contrast with the other approaches which use only variables (functional style). Both allow more natural expression of the notions and of the algorithms, as well as more efficient proofs. The use of multisets and of patterns leads to a specific collection of techniques which are original and specific to our proof methods [9]. Our work focuses on the practical integration of the various proof techniques and transformation methods in one system, *Theorema*, which allows the construction of the domain theory, the automation of the proofs and of the program transformation steps, as well as the actual testing of the algorithms. In this paper we present for the first time our experiments of algorithm transformation: introducing tail recursion, adding a useful flag, pattern–based to functional, functional to imperative.

II. Context

A. Notations

For function and for predicate application we use square brackets, for instance: $f[x]$ instead of $f(x)$ and $P[a]$ instead of $P(a)$. Quantified variables are placed under the quantifier, as in $\underset{X}{\forall}$ and $\underset{X}{\exists}$.

The objects occurring in the formulae are: *elements* — objects from a totally ordered domain (denoted a, b, c) which are members of composite objects; *multisets* denoted A, B, C; and *lists* denoted U, V, W, X, Y, Z. Multisets and lists are addressed as *composite objects*. The ordering on the basic domain is extended to composite objects, by requiring that all the elements observe the relation. The type information is not addressed explicitly in the proof, but it is used implicitly based on the notation conventions: lower case variables and constants represent domain elements, while upper case refers to lists. The same convention applies to function and predicate names, when they are not denoted by special symbols. Additionally the prover uses implicitly certain basic properties of ordering, of multiset union, etc.

The multiset of a list observes:

Property 1. $$\underset{a,U}{\forall}\left(\begin{array}{c}\mathcal{M}[\langle\rangle]=\emptyset\\ \mathcal{M}[a\frown U]=\{\!\{a\}\!\}\uplus\mathcal{M}[U]\end{array}\right)$$

Sorted lists have the definition:

Definition 1.
$$\underset{a,U}{\forall}\left(\begin{array}{c}\mathit{IsSorted}[\langle\rangle]\\ \mathit{IsSorted}[U\frown a]\Longleftrightarrow(U\leq a\wedge\mathit{IsSorted}[U])\end{array}\right)$$

B. The problem

The problem specification consists in: given a list find its sorted version.

Starting from the specification we build the logical conjecture as an input condition $I[X]$ and an output condition $O[X, Y]$.

For univariate functions the conjecture is:

Conjecture 1. $\underset{X}{\forall}(I[X]\implies\underset{Y}{\exists}O[X,Y])$.

For a bivariate function one has $I[X, Y]$, $O[X, Y, Z]$, and the conjecture:

Conjecture 2. $\underset{X,Y}{\forall}(I[X,Y]\implies\underset{Z}{\exists}O[X,Y,Z])$.

C. Proof Techniques

We use the inference rules and strategies introduced in [9], which benefit from the use of multisets.

IR-1: *Forward inference.* Nonclausal unit resolution between assumptions. Example: used to prove (11) on the basis of (12).

IR-2: *Backward inference.* Nonclausal resolution between an unit assumption and the goal. Example: (3) to (4).

IR-3: *Reduce composite argument.* Split an atom containing composite arguments into atoms containing simple arguments (variables and constants). Example: (6) to (7).

IR-4: *Solve metavariable.* Obtain the value of a metavariable from an equality of multiset terms. Example: (25).

IR-5: *Expand multiset.* Transform a multiset term into several terms. Example: (20) to (21).

IR-6: *Compress multiset.* Group two multiset terms into one. Example: (23) to (24).

IR-7: *Use equivalence.* Use the equivalence (induced by the equality of multisets) for transforming an atom. Example: (10) to (11).

IR-8: *Two constants.* Generate cases w.r.t. the ordering of two constants when a second constant element is introduced in the proof. Example: after (21).

ST-1: *Cover set.* Generate proof alternatives using a cover set for a constant or for a metavariable. Example: applies on (2) using a cover set for a metavariable and generates two branches *Case 1.1* and *Case 1.2*.

ST-2: *Induction.* Generate an induction hypothesis and a recursive term in the goal for a term which is smaller (with respect to the Noetherian metaordering induced by the strict inclusion of multisets) than the current target constant. Example: (9).

ST-3: *Cascading.* Generate a conjecture for the synthesis of a necessary auxiliary function, the appropriate property and the appropriate term in the current goal. Example: from goal (11) using (5) generate **Conjecture 4**.

ST-4: *Pair multisets.* For a pair of multiset terms in the goal, generate an equal term, by using a known property.

ST-5: *Split.* For a multiset term in the goal, generate an equal pair of terms, by using a known property.

ST-6: *Split goal equation.* Use heuristics to split a goal equations which contains two metavariables into equations containing only one metavariable each. Example: (23) to (25).

III. Synthesis of *Max-Sort*

The synthesis conjecture is:

Conjecture 3. $\underset{XV}{\forall\exists}(\mathcal{M}[V]=\mathcal{M}[X]\ \wedge\ \mathit{IsSorted}[V])$.

The proof is similar to Proof 1 from [9] which synthesizes *Min-Sort* except the second case, where the cover set to V^* determined by the definition is $U^*\frown a^*$.

Proof 1: Universal X is Skolemized to the *target constant* X_0, producing the *target goal*:

$$\underset{V}{\exists}\mathcal{M}[V] = \mathcal{M}[X_0] \ \wedge \ \mathit{IsSorted}[V] \tag{1}$$

and the existential V becomes the metavariable V^*:

$$\mathcal{M}[V^*] = \mathcal{M}[X_0] \ \wedge \ \mathit{IsSorted}[V^*]. \tag{2}$$

Strategy **ST-1** is applied to the metavariable V^* using the cover set: $\{\langle\rangle,\ U^* \frown a^*\}$:
Case 1.1. $V^* = \langle\rangle$. The goal (2) becomes:

$$\mathcal{M}[\langle\rangle] = \mathcal{M}[X_0] \ \wedge \ \mathit{IsSorted}[\langle\rangle]. \tag{3}$$

By inference rule **IR-2** (backward inference) using **Definition 1** the goal (3) becomes:

$$\mathcal{M}[\langle\rangle] = \mathcal{M}[X_0]. \tag{4}$$

By **ST-1** the proof succeeds on this branch, the witness is $\langle\rangle$, the condition on the input is $X = \langle\rangle$, and the cumulated condition on the input for the next branch is $X_0 \neq \langle\rangle$.
Case 1.2 $V^* = U^* \frown a^*$. The condition on X_0 from the previous branch is added as assumption:

$$X_0 \neq \langle\rangle. \tag{5}$$

The goal (2) becomes:

$$\mathcal{M}[U^* \frown a^*] = \mathcal{M}[X_0] \ \wedge \ \mathit{IsSorted}[U^* \frown a^*] \tag{6}$$

and the current solution for V^* is $U^* \frown a^*$. By inference rule **IR-3** (reduce composite argument) using **Definition 1** the goal (6) becomes:

$$\mathcal{M}[U^* \frown a^*] = \mathcal{M}[X_0] \ \wedge \ U^* \leq a^* \ \wedge \ \mathit{IsSorted}[U^*]. \tag{7}$$

By **IR-2** (backward inference) U^* is replaced by $\mathit{Sort}[W^*]$, the goal becomes:

$$\begin{array}{c}\mathcal{M}[\mathit{Sort}[W^*] \frown a^*] = \mathcal{M}[X_0] \ \wedge \\ \mathit{Sort}[W^*] \leq a^* \ \wedge \ \mathit{IsSorted}[\mathit{Sort}[W^*]]\end{array} \tag{8}$$

and the intermediate solution for V^* is $\mathit{Sort}[W^*] \frown a^*$. Since $\mathit{Sort}[W^*] \frown a^*$ stands for V^* which has the same elements as the target constant X_0, the prover infers that W^* is less than X_0 in the well–founded ordering, thus by strategy **ST-2** (induction) the target goal (1) is used with $\{X \to W^*\}$ and $\{V \to \mathit{Sort}[W^*]\}$ to generate the assumption:

$$\mathcal{M}[\mathit{Sort}[W^*]] = \mathcal{M}[W^*] \ \wedge \ \mathit{IsSorted}[\mathit{Sort}[W^*]]. \tag{9}$$

The second conjunct of this assumption is used to reduce the goal (8) by rule **IR-2** to:

$$\mathcal{M}[\mathit{Sort}[W^*] \frown a^*] = \mathcal{M}[X_0] \ \wedge \ \mathit{Sort}[W^*] \leq a^*. \tag{10}$$

The first conjunct is used by **IR-7** (use equivalence) to reduce the last goal to:

$$\mathcal{M}[W^* \frown a^*] = \mathcal{M}[X_0] \ \wedge \ W^* \leq a^*. \tag{11}$$

The strategy **ST-3** (cascading) is applied to this goal and generates the conjecture:

Conjecture 4.
$\underset{X}{\forall}(X \neq \langle\rangle \Longrightarrow \underset{a\,U}{\exists\exists}(\mathcal{M}[U \frown a] = \mathcal{M}[X] \ \wedge \ U \leq a))$.

Proof 2 synthesizes the functions $\mathit{max}[X]$ and $\mathit{Trimm}[X]$ which split a list into its maximum and the rest.

By **ST-3** (cascading) the new assumption is:

$$\begin{array}{r}\underset{X}{\forall}(X \neq \langle\rangle \Longrightarrow (\mathcal{M}[\mathit{Trimm}[X] \frown \mathit{max}[X]] = \mathcal{M}[X] \ \wedge \\ \mathit{Trimm}[X] \leq \mathit{max}[X])).\end{array} \tag{12}$$

Using (5) this solves the goal (11) with the witnesses: $\{a^* \to \mathit{max}[X_0]\}$ and $\{W^* \to \mathit{Trimm}[X_0]\}$, which gives for V^* the final solution $\mathit{Sort}[\mathit{Trim}[X_0]] \frown \mathit{max}[X_0]$.

QED

The extracted algorithm is:

Algorithm 1. Max-Sort.

$$\left(\begin{array}{c}\mathit{MSort}[\langle\rangle] = \langle\rangle \\ U \neq \langle\rangle \Longrightarrow \mathit{MSort}[U] = \mathit{MSort}[\mathit{Trimm}[U]] \frown \mathit{max}[U]\end{array}\right)$$

A. Synthesis of max and Trimm

The target functions are $\mathit{max}[X]$ which selects from X the maximum element according to the domain ordering and $\mathit{Trimm}[X]$ which gives the list without it. We need to prove **Conjecture 4**.

Proof 2: By natural style proving, take X_0 arbitrary but fixed, assume:

$$X_0 \neq \langle\rangle \tag{13}$$

and after introducing the existential metavariables, the goal is:

$$\mathcal{M}[X_0] = \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \ \wedge \ Y^* \leq y^*. \tag{14}$$

By **IR-7** (use equivalence) the goal is transformed into:

$$\mathcal{M}[X_0] = \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \ \wedge \ X_0 \leq y^*. \tag{15}$$

Strategy **ST-1** (cover set) applies to X_0, using only $U_0 \frown a_0$ because (13). The goal is:

$$\mathcal{M}[U_0 \frown a_0] = \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \ \wedge \ a_0 \frown U_0 \leq y^*. \tag{16}$$

By **IR-3** (composite argument) on the last conjunct the goal becomes:

$$\mathcal{M}[U_0 \frown a_0] = \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \wedge a_0 \leq y^* \wedge U_0 \leq y^*. \tag{17}$$

Strategy **ST-3** (cascading) generates the conjecture:

Conjecture 5.
$\underset{X\,a\,y\,Y}{\forall\forall\exists\exists}(\mathcal{M}[X \frown a] = \mathcal{M}[Y] \uplus \{\!\{y\}\!\} \ \wedge \ a \leq y \ \wedge \ X \leq y)$.

Proof 3 synthesizes the auxiliary functions *maxA* and *TrimmA* which have the property:

$$\begin{array}{r}\underset{X\,a}{\forall\forall}(\mathcal{M}[X \frown a] = \mathcal{M}[\mathit{TrimmA}[X,a]] \uplus \{\!\{\mathit{maxA}[X,a]\}\!\} \wedge \\ a \leq \mathit{maxA}[X,a] \ \wedge \ X \leq \mathit{maxA}[X,a])\end{array} \tag{18}$$

and which solves the goal (17) using the witnesses $\{Y^* \to \mathit{TrimmA}[U_0, a_0],\ y^* \to \mathit{maxA}[U_0, a_0]\}$.

QED

We prove now **Conjecture 5** for the synthesis of *maxA* and *TrimmA*.

Proof 3: By quantified inferences the goal becomes:

$$\mathcal{M}[X_0 \frown a_0] = \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \wedge a_0 \leq y^* \wedge X_0 \leq y^*. \quad (19)$$

Strategy **ST-1** applies to X_0 with the cover set $\{\langle\rangle,\ b_0 \backsim U_0\}$.
Case 1. $X_0 = \langle\rangle$ is straightforward, the solutions are: $\{y^* \to a_0,\ Y^* \to \langle\rangle\}$.
Case 2. $X_0 = b_0 \backsim U_0$ generates the goal:

$$\begin{aligned}\mathcal{M}[a_0 \backsim (b_0 \backsim U_0)] = \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \wedge \\ a_0 \leq y^* \ \wedge\ b_0 \backsim U_0 \leq y^*.\end{aligned} \quad (20)$$

By **IR-5** (expand multiset) and **IR-3** (reduce composite argument) the goal becomes:

$$\begin{aligned}\{\!\{a_0\}\!\} \uplus \{\!\{b_0\}\!\} \uplus \mathcal{M}[U_0] = \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \wedge \\ a_0 \leq y^* \ \wedge\ b_0 \leq y^* \ \wedge\ U_0 \leq y^*.\end{aligned} \quad (21)$$

Two cases for domain element constants are generated by rule **IR-8** (two constants):
Case 2.1. $a_0 \leq b_0$. Strategy **ST-2** (induction) applies to U_0, b_0 in (19) and adds the assumption:

$$\begin{aligned}\mathcal{M}[U_0] \uplus \{\!\{b_0\}\!\} = \mathcal{M}[\mathit{TrimmA}[U_0, b_0]] \uplus \{\!\{\mathit{maxA}[U_0, b_0]\}\!\} \ \wedge \\ b_0 \leq \mathit{maxA}[U_0, b_0] \ \wedge U_0 \leq \mathit{maxA}[U_0, b_0].\end{aligned} \quad (22)$$

(21) is rewritten by equality (22):

$$\begin{aligned}\mathcal{M}[\mathit{TrimmA}[U_0, b_0]] \uplus \{\!\{\mathit{maxA}[U_0, b_0]\}\!\} \uplus \{\!\{a_0\}\!\} = \\ \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \ \wedge \\ b_0 \leq y^* \quad \wedge \quad a_0 \leq y^* \quad \wedge \quad U_0 \leq y^*.\end{aligned} \quad (23)$$

Inference rule **IR-6** composes a multiset from $\{\!\{a_0\}\!\}$ and $\mathcal{M}[\mathit{TrimmA}[U_0, b_0]]$ transforming the goal into:

$$\begin{aligned}\mathcal{M}[a_0 \backsim \mathit{TrimmA}[U_0, b_0]] \uplus \{\!\{\mathit{maxA}[U_0, b_0]\}\!\} = \\ \mathcal{M}[Y^*] \uplus \{\!\{y^*\}\!\} \ \wedge \\ b_0 \leq y^* \quad \wedge \quad a_0 \leq y^* \quad \wedge \quad U_0 \leq y^*.\end{aligned} \quad (24)$$

The goal equation is split by strategy **ST-6**:

$$\begin{aligned}\mathcal{M}[\mathit{TrimmA}[U_0, b_0]] \uplus \{\!\{a_0\}\!\} = \mathcal{M}[Y^*] \wedge \\ \{\!\{\mathit{maxA}[U_0, b_0]\}\!\} = \{\!\{y^*\}\!\} \ \wedge\ b_0 \leq y^* \ \wedge\ a_0 \leq y^* \ \wedge\ U_0 \leq y^*.\end{aligned} \quad (25)$$

By **IR-4** (solve metavariable) the solutions are: $\{y^* \to \mathit{maxA}[U_0,],\ Y^* \to a_0 \backsim \mathit{TrimmA}[U_0, b_0]\}$ and the remaining goal is proven by standard logic and properties of ordering.
Case 2.2. $b_0 < a_0$. The proof proceeds similarly by applying induction on U_0, a_0 in (19) and the obtained solutions are: $\{y^* \to \mathit{maxA}[U_0, a_0],\ Y^* \to b_0 \backsim \mathit{TrimmA}[U_0, a_0]\}$.
QED

The extracted algorithms from the proofs are[3]:

[3]In the presentation of the algorithms it is assumed that all variables are universally quantified over their respective domains, according to our notation convention.

Algorithm 2. Maximum.

$$\begin{pmatrix} \mathit{max}[a \backsim U] = \mathit{maxA}[U, a] \\ \mathit{maxA}[\langle\rangle, a] = a \\ \mathit{maxA}[b \backsim U, a] = \begin{cases} \mathit{maxA}[U, b], & \mathit{if}\ a \leq b \\ \mathit{maxA}[U, a], & \mathit{if}\ b < a \end{cases} \end{pmatrix}$$

Algorithm 3. Trimm.

$$\begin{pmatrix} \mathit{Trimm}[a \backsim U] = \mathit{TrimmA}[U, a] \\ \mathit{TrimmA}[\langle\rangle, a] = \langle\rangle \\ \mathit{TrimmA}[b \backsim U, a] = \begin{cases} a \backsim \mathit{TrimmA}[U, b], & \mathit{if}\ a \leq b \\ b \backsim \mathit{TrimmA}[U, a], & \mathit{if}\ b < a \end{cases} \end{pmatrix}$$

IV. Synthesis of Bubble Sort

The use of the two functions *max* and *Trimm* together is quite inefficient: the scan of the list is performed twice, using the same test at each step. In order to solve this we propose a systematic method for transforming the functions into a tail recursive version, and then to combine them into a single function. The function *max* is already tail recursive. The function *TrimmA* can be made tail recursive by adding a third argument as "accumulator" for the result. However, for constructing this accumulator, one needs to reverse the operation of the original algorithm: if in *TrimmA* the result is constructed by $\backsim$ (*cons*), in the tail recursive version it must be constructed by the dual operation $\frown$ (*append*) which places an element at the end of a list. For efficiency, this program needs a running environment in which *append* is implemented as constant time operation. Moreover, now the two functions can be merged into one, which returns the pair of maximum and the list without it:

Algorithm 4. Tail recursive max and Trimm.

$$\begin{pmatrix} \mathit{maxTrimm}[a \backsim U] = \mathit{maxTrA}[U, a, \langle\rangle] \\ \mathit{maxTrA}[\langle\rangle, a, V] = \langle V, a\rangle \\ \mathit{maxTrA}[b \backsim U, a, V] = \begin{cases} \mathit{maxTrA}[U, b, V \frown a], & \mathit{if}\ a \leq b \\ \mathit{maxTrA}[U, a, V \frown b], & \mathit{if}\ b < a \end{cases} \end{pmatrix}$$

The nontrivial branch of the sorting algorithm can now be expressed in the following way, also as a tail recursive function, which is in fact the algorithm *Bubble-Sort*:

Algorithm 5. Bubble-Sort.

$$\begin{pmatrix} \mathit{BSort}[a \backsim U] = \mathit{BSortA}[\mathit{maxTrA}[U, a, \langle\rangle], \langle\rangle] \\ \mathit{BSortA}[\langle\langle\rangle, a\rangle, V] = a \backsim V \\ \mathit{BSortA}[\langle b \backsim U, a\rangle, V] = \mathit{BSortA}[\mathit{maxTrA}[U, b, \langle\rangle], a \backsim V] \end{pmatrix}$$

This algorithm is known as its more efficient version which finishes as soon as the list is already sorted.

For this improvement one needs to detect the situations in which the list does not change by application of *MSort*, that is: $\mathit{MSort}[W] = W$, which can be rewritten using *Front* and *last* which decompose a list, as:

$$\mathit{MSort}[\mathit{Trimm}[W]] \frown \mathit{max}[W] = \mathit{Front}[W] \frown \mathit{last}[W] \quad (26)$$

From this follow $max[W] = last[W]$ and $Trimm[W] = Front[W]$. The later can be inductively expressed as

$$Trimm[b \frown V] = Front[b \frown V], \tag{27}$$

which is true when V is empty, and otherwise one has

$$TrimmA[b \frown U, a] = Front[b \frown (a \frown U)] = b \frown Front[a \frown U]. \tag{28}$$

In the definition of *TrimmA* we can see that the clause in which the result begins with b is prefixed by the condition $a \leq b$, namely the value $b \frown TrimmA[a \frown U]$. Thus this condition must hold in order that the list remains unchanged by a new call to *BSort*, and in this case the equality reduces to $TrimmA[a \frown U] = Front[a \frown U]$, which the inductively reduced version of (27). Therefore the relation $a \leq b$ must hold at *every* recursive call of *maxA*, *TrimmA*, and *maxTrA*. Therefore we introduce an additional output value to *maxTrA*, namely a *flag* f which is initialized with T and switches to F when the condition $a \leq b$ does not hold. The new algorithm is:

Algorithm 6. Tail recursive max and Trimm with flag.

$$\begin{pmatrix} maxTrimm[a \frown U] = maxTrA[U, a, \langle\rangle, T] \\ maxTrA[\langle\rangle, a, V, f] = \langle V, a, f\rangle \\ maxTrA[b \frown U, a, V, f] = \begin{cases} maxTrA[U, b, V \frown a, f], \text{ if } a \leq b \\ maxTrA[U, a, V \frown b, F], \text{ if } b < a \end{cases} \end{pmatrix}$$

The sorting algorithm will stop recursion as soon as the flag remains true, but for forming the result it must use the function *Conc* which concatenates two lists – for efficiency we assume that the running environment implements this as a constant time operation.

Algorithm 7. Bubble-Sort with flag.

$$\begin{pmatrix} BSort[a \frown U] = BSortA[maxTrA[U, a, \langle\rangle, T], \langle\rangle] \\ BSortA[\langle U, a\rangle, V, T] = Conc[U, a \frown V] \\ BSortA[\langle b \frown U, a\rangle, V, F] = BSortA[maxTrA[U, b, \langle\rangle, T], a \frown V] \end{pmatrix}$$

The use of the flag is crucial for the efficiency of the bubble-sort algorithm, because this algorithm, although is not of optimal efficiency, still is preferable for the re-sorting of lists which have changed slightly since being already sorted. This is used for instance in graphic processing, in order to sort the surfaces by the distance from the viewing point, when its position changes.

V. Transformation into an Imperative Program

The advantage of a tail recursive algorithm is the possibility of transforming it into an imperative program using loops instead of recursion, which is usually much more efficient, especially in the case of loops with a relatively low number of operations, like in the sorting algorithm.

We proceed in two steps, first we generate the *functional* program, and then the imperative one.

For the functional program one uses the functions which are reverse to the pattern $a \frown U$ used in the algorithms, namely *head* and *Tail*. We also use the construct *Let*[4], which consists in creating some variables and assigning them certain values, and then returning the value of the last expression in the construct. In *Let* we allow the use of assignments in tuple form: a tuple of variables is assigned a tuple of values, with the meaning that each variable is assigned the corresponding value in the other tuple. This assignment takes place in the same time for all the variables (similar to the situation of substitutions, or of *Let* from *Lisp*).

Algorithm 8. Functional max and Trimm with flag.

$$\begin{pmatrix} maxTrimmF[U] = maxTrAF[Tail[U], head[U], \langle\rangle, T] \\ maxTrAF[\langle\rangle, a, V, f] = \langle V, a, f\rangle \\ maxTrAF[U, a, V, f] = Let[b = head[U], W = Tail[U], \\ \qquad \begin{cases} maxTrAF[W, b, V \frown a, f], \text{ if } a \leq b \\ maxTrAF[W, a, V \frown b, F], \text{ if } b < a \end{cases}] \end{pmatrix}$$

Algorithm 9. Functional Bubble-Sort with flag.

$$\begin{pmatrix} BSortF[U] = BSortAF[maxTrAF[Tail[U], head[U], \langle\rangle, T], \langle\rangle] \\ BSortAF[\langle U, a\rangle, V, T] = \\ \begin{cases} Conc[U, a \frown V], & \text{if } f \\ BSortAF[maxTrAF[Tail[U], head[U], \langle\rangle, T], a \frown V], & ow. \end{cases} \end{pmatrix}$$

The functional programs are transformed into imperative in a straightforward way: each argument of the recursive is represented by a certain variable, which is at the beginning initialized with the value from the call, and then it is updated at each loop according to the functional algorithm. The loop is a *While* on the negated condition for the return of the final value, which thus repeats the updates of the variables, for which we use again a simultaneous assignment of a vector[5] of values to a vector of variables. After the loop ends, *Return* gives the final result of the call. The algorithms are presented in Fig. 1.

VI. Conclusion and Further Work

Our experiments demonstrate the possibility of generating *Bubble-Sort* with intuitive and short proofs, in which induction and synthesis of auxiliary functions are relatively straightforward. Moreover, we expose systematic principles for transformation of the algorithms into tail–recursive form, to improve the efficiency using a special flag, and even more by transformation into functional and iterative programs.

Generation of a similar algorithm starting from *Min-Sort* [9] is very similar, and leads essentially to the same algorithm. This encourages to attempt the synthesis of the *improved Bubble-Sort*[6] which selects at every list scan both the minimum and the maximum elements and to compare the efficiency.

Generation of *Bubble-Sort* on the path followed by [18] is also possible by the same principles, however this approach is based on moving the maximum to the last position (*MaxL*),

[4]Like in *Lisp*.

[5]Vectors, as well as the empty list, are denoted using square brackets.

[6]https://www.academia.edu/25304876/Comparison_of_Bubble_Sort_and_Selection_Sort_with_their_Enhanced_Versions

Algorithm 10. Imperative max and Trimm with flag.

```
maxTrAI(Win, ain, Vin, fin){
  local W, a, V, f, b, U;
  [W, a, V, f] := [Win, ain, Vin, finb];
  While(W != []){
    b := head(W); U := Tail(W);
    [W, a, V, f] := If(a <= b)
       then[U, b, append(V, a), f]
       else[U, a, append(V, b), False];
  };
  Return [V, a, f];
};
```

Algorithm 11. Imperative Bubble-Sort with flag.

```
BSortI(W){
  Return If(W = [])then[]
    else
    BSortAI(
      maxTrAI(Tail(W), head(W), [], True),
      []);
};
BSortAI(triplein, Vin){
  local triple, V, U, a, f;
  [triple, V] := [triplein, Vin];
  [U, a, f] := triple;
  While(! f){
    [triple, V] :=
      [maxTrAI(Tail(U), head(U), [], True),
       cons(a, V)];
    [U, a, f] := triple;
  };
  Return concat(U, cons(a, V))
};
```

Fig. 1. Iterative algorithms.

finding the last element (*last*), and removing it (*DelL*), which appears to be more cumbersome an less suitable for improvement:

Algorithm 12. Bubble-Sort.

$$\begin{pmatrix} F_2[\langle\rangle] = \langle\rangle \\ U \neq \langle\rangle \Longrightarrow F_2[U] = F_2[\mathit{DelL}[\mathit{MaxL}[U]]] \frown \mathit{last}[\mathit{MaxL}[U]] \end{pmatrix}$$

Other possibilities for further experiments are the automatic synthesis and transformation of other sorting algorithms, and the automatic evaluation of their efficiency.

REFERENCES

[1] D. R. Barstow. Remarks on "A Synthesis of Several Sorting Algorithms" by John Darlington. *Acta Informatica*, 13:225–227, 1980.

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

[3] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, 2005.

[4] R. M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[5] J. Darlington. *A semantic approach to automatic program improvement*. PhD thesis, University of Edinburg, 1972.

[6] J. Darlington. A Synthesis of Several Sorting Algorithms. *Acta Informatica*, 11:1–30, 1978.

[7] J. Darlington and R. M. Burstall. A System Which Automatically Improves Programs. *Acta Informatica*, 6(1):41–60, 1976.

[8] I. Dramnesc and T. Jebelean. Synthesis of List Algorithms by Mechanical Proving. *Journal of Symbolic Computation*, 68:61–92, 2015.

[9] I. Dramnesc and T. Jebelean. Proof–Based Synthesis of Sorting Algorithms Using Multisets in *Theorema*. In *FROM 2019*, pages 76–91. EPTCS 303, 2019.

[10] I. Dramnesc, T. Jebelean, and S. Stratulat. Mechanical Synthesis of Sorting Algorithms for Binary Trees by Logic and Combinatorial Techniques. *Journal of Symbolic Computation*, 90:3–41, 2019.

[11] R. Geoff Dromey. Derivation of Sorting Algorithms from a Specification. *Computer Journal*, 30(6):512–518, 1987.

[12] Brian T. Howard. Another iteration on "A synthesis of several sorting algorithms", 1994.

[13] Yomna Ben Jmaa, Rabie Ben Atitallah, David Duvivier, and Maher Ben Jemaa. A Comparative Study of Sorting Algorithms with FPGA Acceleration by High Level Synthesis. *Computación y Sistemas*, 23:213, 2019.

[14] Yulia Korukhova. An Approach to Automatic Deductive Synthesis of Functional Programs. *Annals of Mathematics and Artificial Intelligence*, 50(3-4):255–271, 2007.

[15] K. K. Lau. Top-down Synthesis of Sorting Algorithms. *The Computer Journal*, 35:A001–A007, 1992.

[16] Yanhong A. Liu and Scott D. Stoller. From Recursion to Iteration: What Are the Optimizations? *SIGPLAN Not.*, 34(11):73–82, 1999.

[17] Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and System*, 2(1):90–121, 1980.

[18] Jonathan Traugott. Deductive Synthesis of Sorting Programs. *Journal of Symbolic Computation*, 7(6):533–572, 1989.

[19] W. Windsteiger. Theorema 2.0: A System for Mathematical Theory Exploration. In *ICMS'2014*, volume 8592 of *LNCS*, pages 49–52, 2014.