Combinatorial Techniques for Proof-Based Synthesis of Sorting Algorithms

Isabela Dramnesc, Tudor Jebelean, Sorin Stratulat

Combinatorial Techniques for Proof–based Synthesis of Sorting Algorithms

Isabela Drămnesc
Department of Computer Science
West University
Timişoara, Romania
Email: idramnesc@info.uvt.ro

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

Sorin Stratulat
LITA, Department of Computer Science
Université de Lorraine
Metz, France
Email: sorin.stratulat@univ-lorraine.fr

***Abstract*—In the frame of our previous experiments for proof based synthesis of sorting algorithms for lists and for binary trees, we employed certain special techniques which are able to generate multiple variants of sorting and merging, by investigating all combinations of auxiliary functions available for composing objects (lists, respectively trees). The purpose of this paper is to describe this technique and the results obtained. We present the main principles and the application of this technique to merging of sorted binary trees into a sorted one. Remarkably, merging requires a nested recursion, for which an appropriate induction principle is difficult to guess. Our method is able to find it automatically by using a general Noetherian induction and the combinatorial technique.**

***Index Terms*—automated reasoning, algorithm synthesis, Theorema, binary trees**

I. Introduction

Synthesizing programs from specifications is a challenging problem [12] that can be tackled by using formal reasoning. Current successful synthesizing formal systems [13] mix automated and interactive developments. Most of their success depends on the way deductive and inductive reasoning is performed and combined. In this paper, we are interested to analyze these aspects when dealing with proof-based synthesis of functional algorithms defined over unbounded data structures like binary trees.

In our setting, the formal specification of a target function $F[\overline{X}]$ (where $\overline{X}$ is a vector of variables) is defined in terms of two conditions given as predicates: $I[\overline{X}]$ (input condition) and $O[\overline{X}, F[\overline{X}]]$ (output condition). The soundness property for F, $\underset{\overline{X}}{\forall}(I[\overline{X}] \Rightarrow O[\overline{X}, F[\overline{X}]])$, ensures that whenever the input of F satisfies I, its input and output satisfy O. The synthesis problem is formalized as the implication $\underset{\overline{X}}{\forall}\underset{T}{\exists}(I[\overline{X}] \Rightarrow O[\overline{X}, T])$ for which the soundness property is a particular case when the variable T is Skolemized to $F[\overline{X}]$. Our approach is proof-based [3] by considering the formalized synthesis problem as a conjecture to be proved. The proof of the conjecture is built using domain-specific inference rules and proof strategies. When T is an unbounded data structure, (a potentially infinite number of) different proofs can be generated, depending on the used induction principles and deductive techniques, as the case-analysis. Finally, from any such a proof, a synthesized algorithm can be automatically extracted using transformation rules, its 'form' depending on the way the proof was built.

In the past, we have experimented with proof-based synthesis of sorting algorithms for lists [9] and binary trees [10], the employed proof techniques allowing us to synthesize a bunch of sorting algorithms. In this paper, we are interested to systematically control their generation and analyze the results, by investigating all combinations of auxiliary functions available for composing objects (lists, respectively trees).

The paper is structured as follows: section II summarizes related work, section III describes the proof-based synthesis method, section IV presents the experiments, and finally section V lists the conclusions and further work.

II. Related Work

There are several methods for algorithm synthesis. A comparison between three methods for recursive program synthesis, namely constructive/deductive synthesis, schema-based synthesis and inductive synthesis, has been done in [1] by synthesizing a common program. Paper [1] complements the survey of logic program synthesis which has been done in [4] and [11]. In the context of constructive synthesis, the approach in this paper describes deductive techniques for synthesizing algorithms operating on binary trees.

In [5], [7], and [9] the authors describe proof–based methods and techniques for synthesizing sorting algorithms operating on lists. A similar proof–based approach has been experimented in [6] and [8] in order to synthesize algorithms operating on *monotone lists* (sorted lists without duplications). This paper also describes methods and techniques which are proof–based, but in contrast, the focus in this paper is on synthesizing binary tree algorithms.

In classical approaches, see [14], the problem of sorting trees is not investigated. This paper introduces combinatorial techniques which apply in the process of synthesizing sorting algorithms and of some auxiliary functions necessary in the sorting algorithms on binary trees.

III. Proof–based Method

In this section we present the algorithm synthesis problem and the main idea of proof-based synthesis techniques which we use.

A. Context

1) Notations: Similar to the *Theorema* style, we use square brackets for function and for predicate application (e.g., f[x] instead of f(x) and P[a] instead of P(a)). Moreover the quantified variables are written under the quantifier, that is $\underset{x}{\forall}$ ("for all x") and $\underset{T}{\exists}$ ("exists T"). Sometimes the place under the quantifier also contains a property of the quantified object. New formulas can be obtained from universally quantified formulas $\underset{\overline{x}}{\forall} F[\overline{x}]$ by using *substitutions* that map subsets of free variables from $F[\overline{x}]$ with terms, of the form $\{x_1 \mapsto t_1, \ldots x_n \mapsto t_n\}$, where $x_1, \ldots, x_n$ are free variables from $F[\overline{x}]$. We denote the application of a substitution σ to a formula F by $F\sigma$ and say that $F\sigma$ is an *instance* of F. An *identity* substitution, generically denoted by σ_{id}, maps any free variable from a formula to itself. The *composition* of two substitutions σ_1 and σ_2 is denoted by $\sigma_1\sigma_2$. Similarly, substitutions can be applied to terms and vector of terms.

The types of the objects are implicit (some objects have type "tree"), by using predicate and function symbols which are not overloaded. Lower-case letters (e.g., a, b, n) represent tree elements, and upper-case letters (e.g., X, T, Y, Z) represent trees. The provers generate meta–variables (denoted usually by starred symbols — e.g., T^*, T_1^*, Z^*) and Skolem constants (e.g., X_0, X_1, a_0).

We consider binary trees over a totally ordered domain. The orderings between elements of the domain are denoted by the usual $\leq$, the ordering between a tree and an element are denoted by: $\preceq$ (e.g., $T \preceq z$ states that all the elements from the tree T are smaller or equal than the element z, $z \preceq T$ states that z is smaller or equal than all the elements from the tree T), and $\ll$ for the ordering between the elements of two trees (e.g., $L \ll R$ states that all the elements from L are smaller or equal than all the elements from R). The constructors for binary trees are: ε for the empty tree, and the triplet $\langle L, a, R\rangle$ for non-empty trees, where L and R are trees and a is the root element.

A tree is a *sorted* (or *search*, or *ordered*) tree if it is either ε or of the form $\langle L, a, R\rangle$ such that i) a is greater or equal than any element of L and smaller or equal than any element of R, and ii) L and R are sorted trees.

Functions: RgM, *LfM*, *Concat*, *Insertion*, *Merge* have the following interpretations, respectively: *RgM*$[\langle L, n, R\rangle]$ returns the last visited element by traversing the tree $\langle L, n, R\rangle$ using the in-order (symmetric) traversal; *LfM*$[\langle L, n, R\rangle]$ returns the first element by traversing the tree $\langle L, n, R\rangle$ using the in-order traversal; *Concat*$[X, Y]$ concatenates X with Y (namely, when X is of the form $\langle L, n, R\rangle$ adds Y as a right subtree of the element *RgM*$[\langle L, n, R\rangle]$); *Insertion*$[n, X]$ inserts an element n in a tree X (if X is sorted, then the result is also sorted); *Merge*$[X, Y]$ combines trees X and Y into a new tree (if X, Y are sorted then the result is also sorted).

Predicates: $\approx$ and *IsSorted* have the following interpretations, respectively: $X \approx Y$ states that X and Y have the same elements with the same number of occurrences (but may have different structures), i.e., X is a *permutation* of Y; *IsSorted*$[X]$ states that X is a sorted tree.

The formal definitions of these functions and predicates are presented in [10]. A formal definition of $\approx$ is not given, however we use the properties of $\approx$ as equivalence implicitly in our inference rules and strategies. In particular, we use in our prover the fact that equivalent trees have the same multiset of elements, which translates into equivalent tree–expressions having the same multiset of constants and variables.

The functions *LfM* and *RgM* do not have a definition for the empty tree, however we assume that:

$$\underset{m}{\forall}\Big(\mathit{RgM}[\varepsilon] \leq m \leq \mathit{LfM}[\varepsilon]\Big).$$

Various properties can be proven (mostly by induction) from these definitions – see [10], and they are necessary for the synthesis proofs. The process of finding the necessary axioms, definitions, and properties which are necessary for synthesis is called *theory exploration* – see also [2], and it constitutes an interesting and challenging process.

2) The Synthesis Problem: The main idea is to prove automatically a synthesis conjecture (which corresponds to the specification of the function to be synthesized) and to extract from this proof the algorithm which implements the function. Of course, proving is the difficult part of this process.

The *specification* of the target function $F[X, Y]$ consists of two predicates (we illustrate here the synthesis principle for function with two arguments, for a different number of arguments the principle is similar): the input condition $I[X], I[Y]$ and the output condition $O[X, Y, T]$, and the correctness property for F is $\underset{X,Y}{\forall}((I[X] \wedge I[Y]) \Rightarrow O[X, Y, F[X, Y]])$. The synthesis problem is expressed by the conjecture: $\underset{X,Y}{\forall}\underset{T}{\exists}((I[X] \wedge I[Y]) \Rightarrow O[X, Y, T])$. Proof-based synthesis consists in proving this conjecture and then extracting the algorithm for the computation of F from this proof. In our case the input condition states that X, Y are trees, and since this is implicit, we do not use it.

In this paper we are interested in the synthesis of the function *Merge*$[L, R]$ (merge two sorted trees into a sorted one). The corresponding conjecture is:

Conjecture 1.

$$\underset{\substack{L,R\\ \mathit{IsSorted}[L],\mathit{IsSorted}[R]}}{\forall}\ \underset{T}{\exists}\Big(\mathit{Concat}[L, R] \approx T \wedge \mathit{IsSorted}[T]\Big)$$

B. Induction Principles and Algorithm Extraction

The illustration of the induction principles and algorithm extraction in this subsection is similar to the one from [5], but the induction principles are adapted for trees and the extracted algorithms are more complex.

The *general* induction principle which we use is the Noetherian induction principle [16]. Consider a domain with a well-founded ordering $\lhd$, and the natural extension of this ordering on terms over the domain ($t \lhd t'$ if this also holds for any domain instance). An induction schema to be applied to a

predicate $\underset{\overline{x}}{\forall} P[\overline{x}]$ defined over a vector of variables $\overline{x}$ is a conjunction of instances of $P[\overline{x}]$ called *induction conclusions* that 'cover' $\underset{\overline{x}}{\forall} P[\overline{x}]$, i.e., for any value $\overline{v}$ from the domain of $\overline{x}$, there is an instance of an induction conclusion $P[\overline{t}]$ that equals $P[\overline{v}]$, where $\overline{t}$ is a vector of terms. An induction schema may attach to an induction conclusion $P[\overline{t}]$, as *induction hypotheses*, any instance $P[\overline{t'}]$ of $\underset{\overline{x}}{\forall} P[\overline{x}]$ as long as $\overline{t'}$ is smaller than $\overline{t}$ w.r.t. the well-founded ordering. The induction conclusions without (resp., with) attached induction hypotheses are *base* (resp., *step*) cases of the induction schema.

In the current presentation we consider the domain of binary trees and we use the *multiset of elements* as the measure of binary trees ($\lhd$ is the strict inclusion of the corresponding multisets). Checking strict ordering $t \lhd t'$ between two terms t, t' representing trees reduces to check strict inclusion between the multisets of symbols (constants and variables except ε) occurring in the terms. This is because the expressions representing trees contain only functions which preserve the multiset of elements in the tree: *Concat, Insertion*, and *Merge*. (This preservation of multisets is a consequence of the properties of these functions, which are also proved automatically in our experiments.)

The following *concrete* induction principle is a direct *term-based* instance of the Noetherian induction principle and corresponds to the appropriate *induction schema*:

Induction-1:

$$\Big(P[\varepsilon] \bigwedge \underset{n,L,R}{\forall} \big((P[L] \wedge P[R]) \Longrightarrow P[\langle L, n, R\rangle]\big)\Big) \Longrightarrow \underset{X}{\forall} P[X]$$

The 'covering' property of the two induction conclusions $P[\varepsilon]$ and $P[\langle L, n, R\rangle]$ is satisfied since any binary tree is either ε or of the form $\langle L, n, R\rangle$. $P[L]$ and $P[R]$ are induction hypotheses attached to $P[\langle L, n, R\rangle]$, and it is very easy to see that their terms are smaller than the one of the induction conclusion.

In our experiments we can use automatically this concrete induction principle for proving P as a unary predicate over binary trees (in the case of *Sort*) and for proving P as a binary predicate — induction on the first argument (in the case of *Merge*).

For instance, in order to synthesize the merge algorithm as a function $F[X, Y]$, we consider the output condition $O[X, Y, T] : \big(\mathit{Concat}[X, Y] \approx T \wedge \mathit{IsSorted}[T]\big)$. The corresponding synthesis conjecture is: $\underset{X,Y}{\forall} \underset{T}{\exists} O[X, Y, T]$ by taking $P[X]$ as $\underset{T}{\exists} O[X, Y, T]$.

We perform induction on the first argument, thus the proof is structured as follows:

Base case: For arbitrary but fixed Y_0 (new Skolem constant), we prove $\underset{T}{\exists} O[\varepsilon, Y_0, T]$. If the proof succeeds to find a ground witness $\Im_1[Y_0]$ (a term depending on Y_0) such that $O[\varepsilon, Y_0, \Im_1[Y_0]]$, then we know that $F[\varepsilon, Y] = \Im_1[Y]$. ($Y$ replaces Y_0 in the witness term.)

Step case: For arbitrary but fixed n, X_0 and Y_0 (new constants), we prove $\underset{T}{\exists} O[\langle L_0, n, R_0\rangle, Y_0, T]$. We assume as induction hypotheses $\underset{T}{\exists} O[L_0, Y_0, T]$ and $\underset{T}{\exists} O[R_0, Y_0, T]$, which are Skolemized by introducing two new constants T_1 and T_2 for each existential T. The existentially quantified variable from the goal becomes the meta–variable T^* (for which we need to find a substitution term). If the proof succeeds to find a witness $T^* = \Im_2[L_0, R_0, Y_0, T_1, T_2]$ (term depending on n, L_0, R_0, Y_0, T_1 and T_2), then we know that $F[\langle L, n, R\rangle, Y] = \Im_2[n, L, R, Y, F[L, Y], F[R, Y]]$. ($T_1$ and T_2 are replaced by $F[L, Y]$ and $F[R, Y]$, respectively.)

This function definition expressed as two equalities can be easily transformed into a functional program by using appropriate decomposition functions which extract the root, the left branch, and the right branch from the tree.

The theoretical basis and the correctness of this proof-based synthesis scheme is well known – see for instance [3].

The method for the synthesis of $Sort$ is similar and simpler.

C. Refining Induction by Combinatorial Techniques and Lazy Reasoning

As shown e.g. in [16], sometimes the concrete induction principle which is used for proving does not succeed. In this case one needs to think about a more powerful principle and iterate the proof attempt. We present here a technique which is able to find automatically and in a lazy way, during the proof, new concrete induction principles which are necessary, and which are instances of the general Noetherian induction principle. This is the case for the synthesis of $Merge$ which is presented in section IV at experiments.

This technique is based on combinatorial principles: we generate all possible terms which are solutions of the metavariable (corresponding to the existential goal), and in these terms we also accept the function symbol to be synthesized, as long as it is applied on arguments which are smaller than the arguments of the main call of the current synthesis step. For instance, during synthesis of $Merge$, in the step case (see above), the main call corresponds to the term $F[\langle L_0, n, R_0\rangle, Y]$. If, during the development of the term, $F[L_0, Y]$ is encountered, we can consider $P[L_0, Y] \Rightarrow P[\langle L_0, n, R_0\rangle, Y]$ as an induction case for the new explicit induction schema associated to $F[X, Y]$.

By lazy induction, the induction hypotheses are 'discovered' by need, during the proof process. Compared to eager induction, where the induction hypotheses are defined by explicit induction schemas before the proof starts, the induction reasoning is improved by eliminating the cases when induction hypotheses are defined but not used, or crucial induction hypotheses are needed but not available because the wrong induction schemas were used. In general, when we want to prove a formula $\underset{\overline{x}}{\forall} F[\overline{x}]$ by lazy induction, where $\overline{x}$ is a vector of variables, we start to instantiate variables from $\overline{x}$, then transform the resulted instances by using deductive rules. The instantiation and deduction steps can be intertwined up to the moment when instances of $F[\overline{x}]$ are encountered. An instance $F[\overline{t}]$ can be used as induction hypothesis for the induction case $F[\overline{x}]\theta$ if $\overline{t}$ is smaller than $\overline{x}\theta$.

The substitution θ, called *cumulative* substitution, is built from the proof. To illustrate its computation, we represent the proof derivation as a tree for which the root node is labeled

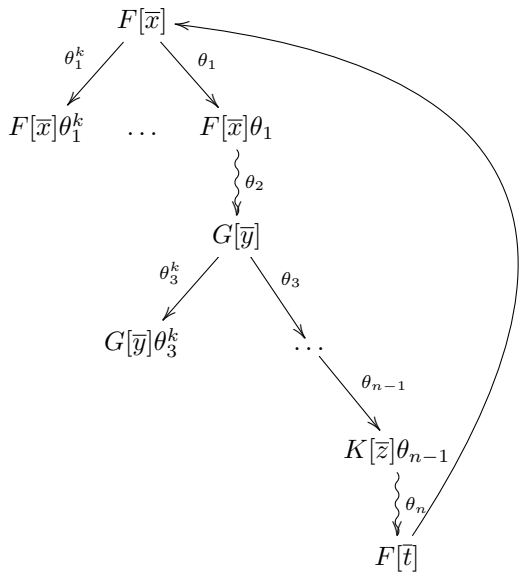
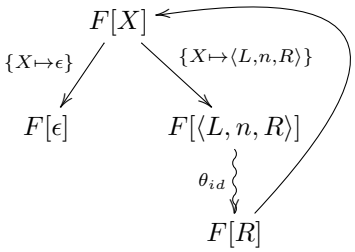


Fig. 1: Applying lazy induction during the proof of $\underset{\overline{x}}{\forall} F[\overline{x}]$.

by $F[\overline{x}]$. Two kinds of non-root nodes are distinguished: instantiation nodes and deductive nodes. The instantiation nodes are direct successors of a node N labeled by a formula with free variables for which some of them are instantiated with terms whose variables are fresh. The set of instance formulas labeling all the instantiation nodes should *cover* the formula labeling N and can be built from the sort of the instantiated variables. For example, if N is labeled by the formula $F[X]$, a covering set of instance formulas is $\{F\{X \mapsto \epsilon\}, F\{X \mapsto \langle L, n, R\rangle\}\}$, where L, n, R are fresh variables. In the graphical representation of a proof tree, the relation between a node and its direct instantiation nodes are represented by downward solid arrows annotated by the corresponding instantiation substitution. The deductive nodes are direct successors of nodes to which a deductive operation has been applied. These relations are graphically represented as curly arrows annotated by identity substitutions. The cumulative substitution is the composition of the substitutions annotating the nodes from the path leading from the root node, in our case the node labeled by $F[\overline{x}]$, to the node labeled by an instance of it, in our case $F[\overline{t}]$. This scenario can be illustrated as in Fig. 1. $F[\overline{t}]$ can be used as an induction hypothesis and its proof no further developed if $\overline{t}$ is smaller than $\overline{x}\theta_1\theta_2\theta_3\cdots\theta_{n-1}\theta_n$.

Example 1. *By lazy induction, one can benefit of more effective induction reasoning, involving only useful induction hypotheses.*

Let us assume the following scenario for processing a formula $F[X]$, where X is a binary tree:

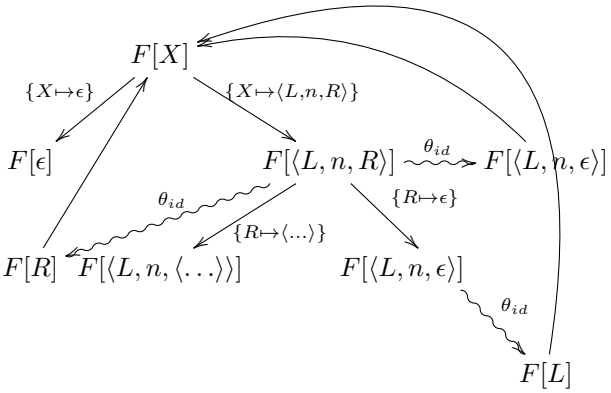


where θ_{id} is the identity substitution $\{L \mapsto L; n \mapsto n; R \mapsto R\}$. $F[R]$ can be used as induction hypothesis in the proof of the case $F[\langle L, n, R\rangle]$ because R has a number of elements smaller than $\langle L, n, R\rangle$.

The corresponding explicit induction principle is:

$$\Big(P[\varepsilon] \bigwedge \underset{n,L,R}{\forall}\big(P[R]) \Longrightarrow P[\langle L, n, R\rangle]\big)\Big) \Longrightarrow \underset{X}{\forall} P[X]$$

Example 2. *More specific induction schemas can also be generated by lazy induction, as shown in the following scenario:*



The corresponding explicit induction principle is:

$$\Big(P[\varepsilon] \bigwedge \underset{n,L}{\forall}\big(P[L] \Longrightarrow P[\langle L, n, \varepsilon\rangle]\big) \bigwedge \underset{n,L,R}{\forall}\big((P[\langle L, n, \varepsilon\rangle] \wedge P[R]) \Longrightarrow P[\langle L, n, R\rangle]\big)\Big) \Longrightarrow \underset{X}{\forall} P[X]$$

Notice that it is a stronger version of ***Induction-1*** *that has been useful for our experiments [10].*

D. Proof techniques

1) Special Inference Rules: We summarize here the main proof principles by which one arrives at the proof situation where the combinatorial techniques can be used. The details on these inference rules and proof strategies are given in [10].

IR-1: *Generate Microatoms.* We call *microatoms* those atoms whose arguments do not contain function symbols, except for few special ones – in the case of the current experiments we allow the functions *RgM* and *LfM* in microatoms.

Based on the specific properties of our functions and predicates, certain atoms can be transformed into a conjunction of microatoms. For instance, *IsSorted*$[\langle T_1, n, T_2\rangle]$ is transformed into ($\mathit{IsSorted}[T_1] \wedge \mathit{IsSorted}[T_2] \wedge \mathit{RgM}[T_1] \leq n \wedge n \leq \mathit{LfM}[T_2]$). Similarly, $x \preceq \langle A, b, C\rangle$ is transformed into:

$x \preceq A \ \wedge x \leq b \wedge \ x \preceq C$.

Some of these microatoms will become conditional assumptions in the synthesized algorithm (see **IR-4** below).

IR-2: *Eliminate-Ground-Formulae-from-Goal.* If the goal contains a ground formula which is identical to (or an instance of) one of the assumptions, then this ground formula is deleted from the goal.

IR-3: *Replace-Equivalent-Term-in-Goal.* If $t_1 \approx t_2$ is an assumption, and t_1 occurs in a goal as argument of a predicate which is preserved by equivalence ($\approx$, $\preceq$), then it can be replaced by t_2. (The fact that these predicates preserve the equivalence is based on their properties, which are proved automatically in our experiments.)

IR-3a: *Replace-Equivalent-Tree-Expression-in-Goal* This generalizes the previous strategy, by constructing a different tree expression which is equivalent.

IR-3b: *Replace-Equivalent-Expression-in-Goal* This rule generalizes **IR-3a**, by allowing similar replacements when the expressions contain function symbols different from the tree constructor. For instance, if among the assumptions are: $\mathit{Concat}[L, S] \approx T_1$, $\mathit{Concat}[R, S] \approx T_2$, and the goal is: $\mathit{Concat}[\langle L, n, R\rangle, S] \approx T^*$, then it is enough to prove one of the two goals: $\langle T_1, n, R\rangle \approx T^*$ and $\langle L, n, T_2\rangle \approx T^*$.

IR-3c: *Replace-Equivalent-Atom-in-Goal* This rule takes into account the interplay between the equivalence relation $\approx$, the orderings, and the functions *RgM*, *LfM* in order to perform similar replacements.

IR-4: *Generate permutations and expressions.*

This is the inference rule which applies the combinatorial techniques which constitute the main focus of the present paper.

When the goal is of the form $\mathit{Expression} \approx T^* \wedge \mathit{IsSorted}[T^*]$, generate all *permutations* of the list of nonempty symbols present in *Expression* and for each permutation generate all possible witnesses as a tree *expressions* containing these symbols. The witnessing expressions contain the constructors for trees, as well as the functions $\mathit{Concat}, \mathit{Insertion}$, and *Merge*. For instance, if *Expression* is ε then only the tree ε is generated. If *Expression* is $\langle L, x, \varepsilon\rangle$, then the generated trees are: $\langle L, x, \varepsilon\rangle$, $\langle \varepsilon, x, L\rangle$, and $\mathit{Insertion}[x, L]$.

Since each such expression must represent a sorted tree, generate for each expression the corresponding *condition* as a set of microatoms (see **IR-1**). For instance the expression $\langle L, x, \varepsilon\rangle$ needs the conditions $\mathit{IsSorted}[L]$ and $L \preceq x$.

Such a condition together with the corresponding expression represents a possible *clause* in the generated algorithm. These clauses are simplified (see the experiments section) according to various criteria, and the remaining set of clauses can be used for the generation of various algorithms, each algorithm being composed of a certain subset of clauses.

Of course if the original expression contains more symbols then the resulting expressions will be more complicated and also quite many – see the experiments in the next section.

Even as this method is used here for the particular case of functions and constructors for trees, the principle and also the implementation is general, it can be applied to any set of functions and constructors. Generating all permutations and all respective expressions appears to be costly, however in our experiments we have a relatively small number of elements and in fact using the special functions of *Mathematica* this takes only few seconds. Therefore the proof is faster than the possible equivalent proof in which the witnesses are generated by logical inference rules in a complex proof tree.

2) Strategies: **S-1:** *Quantifier reduction.* This strategy organizes the inference rules for quantifiers (see **IR-1**), in situations where it is clear that several such rules are to be performed in sequence (e.g. when applying an induction principle).

S-2: *Priority-of-Local-Assumptions.* The *local assumptions* are the assumptions (usually ground formulae) generated during the current proof, and therefore only "true" in the context of the proof. The *global assumptions* (usually universally quantified formulae) are definitions and propositions that are part of the theory, and therefore always "true". The strategy consists in using with priority the local assumptions, and in particular never performing an inference which involves only global assumptions. This strategy is essentially equivalent with the "set of support" strategy in clausal resolution [15].

IV. Experiments

We present here the synthesis of *Merge*, which is more interesting because it needs a double recursion. and therefore it illustrates better the process of finding automatically a concrete induction principle. The proofs corresponding to the synthesis of sorting and other functions are similar, but simpler. For lack of space we show in detail only the most important steps of this proof, the complete version and the other proofs can be found in [10].

As theoretically known, the time complexity of the process is exponential, because it involves search in a proof tree, however in our experiments the concrete running time for all proofs remains under 5 seconds. This is due to the specific inference rules and strategies which are used, because these simulate in one step many low–level (resolution like) inference steps and eliminate many failing branches.

A. Synthesis of Merge

The prover automatically generates the proof of Conjecture 1, which is presented in detail in [10]. We present here only the most important parts of the *induction step* which illustrate the combinatorial technique and the lazy induction.

The proof applies **Induction-1** on the first argument of the function *Merge* to be synthesized. Using strategy **S-1**, after Skolemization of the existential variables into T_1, T_2, the induction hypotheses become:

$$P[L] : \Big((\mathit{IsSorted}[L] \wedge \mathit{IsSorted}[S]) \Longrightarrow (\mathit{Concat}[L, S] \approx T_1 \wedge \mathit{IsSorted}[T_1])\Big) \tag{1}$$

$$P[R] : \Big((\mathit{IsSorted}[R] \wedge \mathit{IsSorted}[S]) \Longrightarrow (\mathit{Concat}[R, S] \approx T_2 \wedge \mathit{IsSorted}[T_2])\Big) \tag{2}$$

and the induction goal (to prove) is:

$$P[\langle L, n, R\rangle] : \Big((\mathit{IsSorted}[\langle L, n, R\rangle] \wedge \mathit{IsSorted}[S]) \Longrightarrow (\mathit{Concat}[\langle L, n, R\rangle, S] \approx T^* \wedge \mathit{IsSorted}[T^*])\Big) \tag{3}$$

where T^* is the metavariable obtained from the existential variable, for which the prover needs to find a witness term. The RHS of the target implication is proven by assuming the LHS, which by **IR-1** is decomposed into microatoms:

$$\mathit{IsSorted}[L] \tag{4}$$
$$\mathit{IsSorted}[R] \tag{5}$$
$$L \preceq n \tag{6}$$
$$n \preceq R \tag{7}$$
$$L \ll R \tag{8}$$
$$\mathit{IsSorted}[S] \tag{9}$$

Using *modus ponens* from (1) and (2) by (4) and (5) further assumptions are obtained:

$$\mathit{Concat}[L, S] \approx T_1 \tag{10}$$
$$\mathit{IsSorted}[T_1] \tag{11}$$
$$\mathit{Concat}[R, S] \approx T_2 \tag{12}$$
$$\mathit{IsSorted}[T_2] \tag{13}$$

The goal is:

$$\mathit{Concat}[\langle L, n, R\rangle, S] \approx T^* \wedge \mathit{IsSorted}[T^*] \tag{14}$$

We need to find a witness for a sorted T^* such that it has the same elements as $Concat[\langle L, n, R\rangle, S]$. (Note that this corresponds to the main call $\mathit{Merge}[\langle L, n, R\rangle, S]$.)

Since **IR-3b** can be applied on (14) in two different ways, we generate two alternatives:

Alternative-1: By applying **IR-3b** using (10), the goal is transformed into:

$$\langle T_1, n, R\rangle \approx T^* \wedge \mathit{IsSorted}[T^*] \tag{15}$$

In this moment the prover uses the *combinatorial technique*, namely it applies **IR-4** and generates all the permutations of $\langle T_1, n, R\rangle$ and to each permutation all possible expressions containing all the symbols in the list, composed by using the tree constructors and the functions *Concat*, *Insertion*, and *Merge*. For each expression the corresponding conditions are generated as a set of microatoms – **IR-1** (except the ones of the form *IsSorted*, which are already known for the tree symbols occurring in the expressions). In this case 42 such clauses are generated.

Some examples of clauses are:

$\{\} \Longrightarrow \mathit{Merge}[T_1, \mathit{Merge}[R, \langle \varepsilon, n, \varepsilon\rangle]]$

$\{\varepsilon \preceq \mathit{LfM}[R]\} \Longrightarrow \mathit{Merge}[T_1, \mathit{Concat}[\langle \varepsilon, n, \varepsilon\rangle, R]]$

$\{\mathit{RgM}[R] \preceq \mathit{LfM}[T_1], \mathit{RgM}[R] \preceq \varepsilon, \mathit{RgM}[R] \preceq n\} \Longrightarrow \mathit{Insertion}[n, \mathit{Merge}[R, T_1]]$

$\{\mathit{RgM}[R] \preceq \mathit{LfM}[T_1], \mathit{RgM}[R] \preceq n, n \preceq \mathit{LfM}[T_1]\} \Longrightarrow \mathit{Concat}[R, \mathit{Concat}[\langle \varepsilon, n, \varepsilon\rangle, T_1]]$

The conditions are simplified by removing the conditions involving ε (which are true by the properties of $\preceq$) and by removing those conditions which are already assumed in the current proof situation.

Furthermore the logical consequences (by transitivity) of the conditions and of the current proof assumptions are computed. If the consequence includes $t \preceq t$ for some term t, this means that both $t \preceq t'$ and $t' \preceq t$ are present for some term t' in the list of conditions and assumptions. This is possible only in very special cases of the application of the algorithm, therefore we remove such clauses. Furthermore we remove from the set of conditions those which are implied by themselves (redundant).

The list of clauses is simplified by removing each clause containing a subterm of the form $\mathit{Merge}[t_2, t_1]$ if the expression of another clause contains $\mathit{Merge}[t_1, t_2]$ in a similar expression at the same level. This because the function *Merge* is symmetric and the conditions are not influenced by the order of its arguments.

The following simplification are also applied because they improve the respective expressions from the computational point of view:

$$\begin{aligned} \mathit{Merge}[\langle \varepsilon, n, \varepsilon\rangle, X] &\longrightarrow \mathit{Insertion}[n, X], \\ \mathit{Merge}[X, \langle \varepsilon, n, \varepsilon\rangle] &\longrightarrow \mathit{Insertion}[n, X], \\ \mathit{Concat}[\langle L, n, \varepsilon\rangle, R] &\longrightarrow \langle L, n, R\rangle. \end{aligned} \tag{16}$$

Each expression is further processed by replacing each occurrence of T_1 by $\mathit{Merge}[L, S]$ — according to (1), and also by replacing the conditions involving T_1 with the appropriate conditions involving L, and S. Finally the duplicate clauses are removed and we obtain a list of 8 clauses (conditions involving *LfM*, *RgM* are presented as simpler equivalent ones for brevity, but the algorithm will of course use them).

$$\{\} \Longrightarrow \mathit{Merge}[\langle \varepsilon, n, R\rangle, \mathit{Merge}[L, S]] \tag{17}$$
$$\{\} \Longrightarrow \mathit{Merge}[\mathit{Insertion}[n, R], \mathit{Merge}[L, S]] \tag{18}$$
$$\{\} \Longrightarrow \mathit{Insertion}[n, \mathit{Merge}[\mathit{Merge}[L, S], R]] \tag{19}$$
$$\{\} \Longrightarrow \mathit{Merge}[\mathit{Insertion}[n, \mathit{Merge}[L, S]], R] \tag{20}$$
$$\{S \preceq n\} \Longrightarrow \langle \mathit{Merge}[L, S], n, R\rangle \tag{21}$$
$$\{S \preceq R\} \Longrightarrow \mathit{Insertion}[n, \mathit{Concat}[\mathit{Merge}[L, S], R]] \tag{22}$$
$$\{S \preceq n\} \Longrightarrow \mathit{Merge}[\langle \mathit{Merge}[L, S], n, \varepsilon\rangle, R] \tag{23}$$
$$\{S \preceq n\} \Longrightarrow \mathit{Concat}[\mathit{Merge}[L, S], \mathit{Insertion}[n, R]] \tag{24}$$

Note that the clauses (17), (18), (20) do not fulfill the termination criterion: the first recursive call to *Merge* has the same multiset of symbols as the main call $\mathit{Merge}[\langle L, n, R\rangle], S]$.

From these clauses various algorithms can be extracted. Each algorithm contains one of the clauses without conditions as the unique or the last clause in the algorithm — but termination is insured only for (19). Additionally the algorithm may contain one of clauses (21), (23), (24), conditioned by $S \preceq n$ and may contain the clause (22) conditioned by $S \preceq R$. Note that the conditioned clauses will lead to more efficient computations (because they have fewer occurrences of the more expensive *Insertion, Merge*), but only when the

conditions of the respective clauses are fulfilled. The choice of the algorithm is therefore a tradeoff between simplicity and efficiency.

One possible algorithm which appears to be a good choice is based on clauses (21), (22), (19) (in this order):

Algorithm 1.

$$\underset{n,L,R,S}{\forall} \left(\begin{array}{c} \mathit{Merge}[\varepsilon, S] = S \\ \mathit{Merge}[\langle L, n, R\rangle, S] = \\ \left\{ \begin{array}{l} \langle \mathit{Merge}[L, S], n, R\rangle, \ \mathit{if}\ \mathit{RgM}[S] \leq n \\ \mathit{Insertion}[n, \mathit{Concat}[\mathit{Merge}[L, S], R]], \\ \qquad\qquad \mathit{if}\ \mathit{RgM}[S] \leq \mathit{LfM}[R] \\ \mathit{Insertion}[n, \mathit{Merge}[\mathit{Merge}[L, S], R]], \ \mathit{otherwise} \end{array} \right. \end{array} \right)$$

The clauses of the algorithm are ordered such that more specific clauses are used before more general ones, because otherwise some more specific clauses will never be used. This ordering is done automatically using the conditions of the clauses: more restrictive conditions belong to more specific clauses.

Alternative-2: By applying **IR-3b** using (12), the goal is transformed into:

$$\langle L, n, T_2\rangle \approx T^* \wedge \mathit{IsSorted}[T^*] \tag{25}$$

The proof proceeds similarly as in *Alternative-1* and similar cases are generated, the only difference consists in using the recursive call $\mathit{Merge}[R, S]$ instead of $\mathit{Merge}[L, S]$ as in Alternative 1. The most important clause generated here is the analogous of (19), namely:

$$\{\} \Longrightarrow \mathit{Insertion}[n, \mathit{Merge}[L, \mathit{Merge}[R, S]]] \tag{26}$$

Further details are presented in the technical report [10].

Any algorithm composed from such conditional clauses must fulfill two important conditions: *proper ordering of clauses* and *coverage of all cases*.

Proper ordering of clauses means that a clause which is more general – like e.g. (19) must be placed after the ones which are less general – like e.g. after (22), otherwise the more special clause will never be used. In our case this is very easily ensured by ordering the clauses increasingly by the number of conditions, because due to the nature of he microatoms, a more general clause always has fewer elements than a more special one. Of course at most one clause with empty condition can be present.

Coverage of all cases means that the disjunctions of all sets of conditions (each set is a conjunction of atoms) must be valid. This is ensured if at least one clause with empty condition is present, and this will always be the case for the merging on binary trees. Validity cannot otherwise be ensured because the induced ordering relations $\preceq$ on elements vs. trees, and $\ll$ on trees vs. trees are not total. However the situation is different in the case of lists – e.g. merging of sorted lists into a sorted one, because there we use as conditions only comparisons between domain elements (not lists). In this case the check of validity can be performed in the following way: (1) each set of conditions is completed with the conditions from the current proof assumptions and with all the transitive consequences; (2) all sets of conditions (as conjunctions) are composed into a disjunction, and its CNF is computed; (3) for validity, each disjunctive clause must be valid, therefore it must contain both $a \leq b$ and $b \leq a$ for some a, b.

In the resulting algorithms the conditions are tested using the functions *LfM, RgM*. In a program with several clauses, multiple calls to these functions can be easily avoided by computing their values before the evaluation of the clauses (the functional `let` from `Lisp`). However their use still remains quite expensive, because the recursive calls will repeat the descending of the tree. This problem (suggested by the automatically generated algorithms) can be solved by changing the data structure: one can store the respective values in each node of the tree (preprocessing for computing them will be linear), and then *LfM, RgM* will be evaluated in constant time.

V. Conclusions and Further Work

We show how to use combinatorial techniques in order to generate numerous witnesses for an existential goal, which in an inductive proof will also lead to some induction schemata which are not given from the beginning. Because of the large number of possible witnesses, a direct proving approach (like e.g. based on clausal resolution) would be very time and space consuming.

From this case study one can see that our method allows to discover new structures of the recursion which are not specified by the induction principle, and moreover allows to discover algorithms with nested recursion.

Correctness of the method is ensured by the soundness of the inference rules, however completeness is still under investigation.

Of course this approach to sorting of binary trees is not really the most efficient — one could always construct first the list of the members and then sort it and construct a tree. This is also further work on the hybrid theory of lists and trees. Nevertheless the exercise in synthesis of sorting and merging the trees directly was very interesting from the point of view of finding efficient proof techniques, and even for suggesting a more efficient data structure (see the end of previous section).

Acknowledgements

Isabela Drămnesc: This work was partially supported by the strategic grant POSDRU/159/1.5/S/137750, Project Doctoral and Postdoctoral programs support for increased competitiveness in Exact Sciences research cofinanced by the European Social Fund within the Sectoral Operational Programme Human Resources Development 2007 – 2013.

References

[1] D. Basin et al. Synthesis of Programs in Computational Logic. In *Program Development in Computational Logic*, pages 30–65. Springer, 2004.

[2] B. Buchberger et al. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[3] A. Bundy et al. Constructing Induction Rules for Deductive Synthesis Proofs. *Electron. Notes Theor. Comput. Sci.*, 153:3–21, 2006.

[4] Y. Deville and K. K. Lau. Logic Program Synthesis. *J. Log. Program.*, 19/20:321–350, 1994.

[5] I. Dramnesc and T. Jebelean. Proof Techniques for Synthesis of Sorting Algorithms. In *Proceedings of SYNASC'11*, pages 101–109. IEEE Computer Society, 2011.

[6] I. Dramnesc and T. Jebelean. Automated Synthesis of Some Algorithms on Finite Sets. In *Proceedings of SYNASC'12*, pages 143–151. IEEE Computer Society, 2012.

[7] I. Dramnesc and T. Jebelean. Discovery of Inductive Algorithms through Automated Reasoning: A Case Study on Sorting. In *Proceedings of SISY'12*, pages 293 – 298. IEEE Xplore, 2012.

[8] I. Dramnesc and T. Jebelean. A Case Study in Proof Based Synthesis of Algorithms on Monotone Lists. In *Proceedings of SACI'15*, pages 483 – 488. IEEE Xplore, 2015.

[9] I. Dramnesc and T. Jebelean. Synthesis of list algorithms by mechanical proving. *Journal of Symbolic Computation*, 68:61–92, 2015.

[10] I. Dramnesc, T. Jebelean, and S. Stratulat. Synthesis of Some Algorithms for Trees: Experiments in Theorema. Technical Report 15-04, RISC Report Series, Johannes Kepler University, Linz, Austria, 2015.

[11] P. Flener. Achievements and Prospects of Program Synthesis. In *Computational Logic: Logic Programming and Beyond*, pages 310–346, 2002.

[12] S. Gulwani. Dimensions in program synthesis. In *Proceedings of PPDP'10*, pages 13–24, New York. USA, 2010. ACM.

[13] E. Kneuss et al. Synthesis Modulo Recursive Functions. In *Proceedings of OOPSLA '13*, pages 407–426, New York, USA, 2013. ACM.

[14] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Addison-Wesley Longman Publishing Co.,Inc., Redwood City, CA, USA, 1998.

[15] R. Kowalski and D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2, 1971.

[16] S. Stratulat. A Unified View of Induction Reasoning for First-Order Logic. In A. Voronkov, editor, *Turing-100 (The Alan Turing Centenary Conference)*, volume 10 of *EPiC Series*, pages 326–352. EasyChair, 2012.