Mechanical Synthesis of Sorting Algorithms for Binary Trees by Logic and Combinatorial Techniques

Isabela Dramnesc, Tudor Jebelean, Sorin Stratulat

Mechanical Synthesis of Sorting Algorithms for Binary Trees by Logic and Combinatorial Techniques

Isabela Drămnesc

West University of Timişoara, Romania

Tudor Jebelean

RISC, Johannes Kepler University, Linz, Austria

Sorin Stratulat

LITA, Université de Lorraine, Metz, France

Abstract

We develop logic and combinatorial methods for automating the generation of sorting algorithms for binary trees, starting from input-output specifications and producing conditional rewrite rules. The main approach consists in proving (constructively) the existence of an appropriate output from every input. The proof may fail if some necessary sub–algorithms are lacking. Then, their specifications are suggested and their synthesis is performed by the same principles.

Our main goal is to avoid the possibly prohibitive cost of pure resolution proofs by using a natural–style proving in which domain-specific strategies and inference steps lead to a significant increase of efficiency. In addition to classical techniques for natural–style proving, we introduce novel ones (priority of certain types of assumptions, transformation of elementary goals into conditions, special criteria for decomposition of the goal and of the assumptions), as well as methods based on the properties of domain-specific relations and functions. In particular, we use combinatorial techniques in order to generate possible witnesses, which in certain cases lead to the discovery of new induction principles. From the proof, the algorithm is extracted by transforming inductive proof steps into recursions, and case-based proof steps into conditionals.

The approach is demonstrated in parallel using the *Theorema* system, by developing the theory, implementing the prover, and performing the proofs of the necessary properties and synthesis conjectures. It is also validated in the *Coq* system, which allows to compare the facilities of the two systems from the point of view of our application.

Keywords: algorithm synthesis, automated reasoning, natural–style proving

Email addresses: isabela.dramnesc@e-uvt.ro (Isabela Drămnesc), Tudor.Jebelean@jku.at (Tudor Jebelean), sorin.stratulat@univ-lorraine.fr (Sorin Stratulat)

URL: http://web.info.uvt.ro/~idramnesc (Isabela Drămnesc), www.risc.jku.at/home/tjebelea (Tudor Jebelean), www.lita.univ-lorraine.fr/~stratula (Sorin Stratulat)

1. Introduction

By **algorithm synthesis** we understand finding an algorithm which satisfies a given specification. We are working in the context of proof-based synthesis of functional algorithms, starting from their formal specifications. A formal specification is expressed as two predicates: the input condition $I[X]$ and the output condition $O[X, T]$.[1] The desired function F must satisfy the correctness condition $\underset{X}{\forall}(I[X] \Rightarrow O[X, F[X]])$, which corresponds to the *synthesis conjecture:* $\underset{X}{\forall}\underset{T}{\exists}(I[X] \Rightarrow O[X, T])$. An algorithm which implements F can be extracted from the (constructive) proof of this conjecture. The algorithm is expressed as a list of *clauses* (conditional rewrite rules) of the form: $C[\overline{Z}] \Longrightarrow F[P[\overline{Z}]] = \mathcal{T}[\overline{Z}]$, where $\overline{Z}$ is a vector of variables, $P[\overline{Z}]$ is a pattern over these variables (with the property that it matches unambiguously certain elements of the domain), and $\mathcal{T}[\overline{Z}]$ is a term over the matching variables. The research presented here is focused on developing effective and efficient techniques for mechanizing the synthesis–proofs of sorting algorithms over the domain of binary trees, the synthesis–proofs of the auxiliary functions occurring in these algorithms, and the proofs of the additional properties which are necessary in the synthesis–proofs.

Our approach is experimental: we try various scenarios and techniques and refine them in order to obtain efficient proofs and various algorithms. The way the constructive proof is built is essential since the definition of the algorithm depends on it. For example, case splits may generate conditional branches and induction steps may produce recursive definitions. Hence, the use of different case reasoning techniques and induction principles may output different algorithms. The soundness of the proof rules and of the extraction procedure guarantee the correctness of the algorithm.

The focus of our work is on proof automation. In our experiments all the proofs are produced completely automatically, while the theory exploration (identification of all necessary auxiliary statements), the selection of the assumptions and of the induction principles used by the prover in each proof, and the construction of the conjectures from the failing proofs are performed manually.

The implementations of the new prover and extractor, as well as of the case studies presented in this paper are carried out in the frame of the *Theorema* system (Buchberger et al., 2016) which is itself implemented in Mathematica (Wolfram, 2003). *Theorema* offers significant support for automating the algorithm synthesis; in particular, the new proof strategies and inference rules have been quickly prototyped, tested and integrated in the system thanks to its extension features. Also, the proofs are easier to understand since they are presented in a human-oriented style. Moreover the synthesized algorithms can be directly executed in the system. The implementation files and the full proofs are presented in (Dramnesc et al., 2015b). We additionally validate the results in the frame of the *Coq* system (Bertot and Casteran, 2004), by mechanically checking that the synthesized sorting algorithms satisfy the correctness conditions.

In parallel with the attempts to prove the conjectures corresponding to the synthesis problems, we explore the theory of binary trees, that is we introduce the necessary axioms and definitions, and we develop the necessary properties and prove them automatically. The full necesary theory is explored in *Theorema*, while in the *Coq* system, we only introduce the notions and properties necessary for the process of certifying the synthesized sorting algorithms.

[1] The square brackets are used for function and predicate applications instead of round brackets.

Our main results include the novel synthesized algorithms: five sorting algorithms plus the auxiliary functions necessary for them: one algorithm for *Insert* (insert an element into a sorted tree), and three algorithms for *Merge* (merge two sorted trees into a sorted one).

More importantly, our experiments reveal a valuable arsenal of proof techniques (inference rules and proof strategies) which are both of general interest in natural–style proving, as well of particular interest in proof–based algorithm synthesis and in proving over the domain of binary trees.

In addition to classical techniques for natural–style proving (induction, Skolemization, meta-variables, goal reduction), we introduce novel ones: priority of certain types of assumptions, transformation of elementary goals into conditions, special criteria for decomposition of the goal and of the assumptions, and other. Additionally we find methods based on the properties of domain-specific relations and functions. In particular, we use combinatorial techniques in order to generate possible witnesses, which in certain cases lead to the discovery of new induction principles.

A specific novel technique is the generation of conditional clauses of the algorithm in case the proof fails on a goal which is simple enough to be considered as a condition. These conditional clauses correspond to `if-then-else` or `case` statements in typical programming languages.

The induction reasoning performed during the proof construction is lazy, meaning that the induction hypotheses are used by need, hence avoiding unnecessary backtracking steps. The lazy induction reasoning can be afterwards captured by explicit induction schemas such that the whole reasoning is reconstructed to an explicit induction proof which is successfully conducted at the end with the *Theorema* system. It may happen that different proof scenarios generate different induction schemas. Some of the auxiliary functions (e.g. *Merge* which merges two sorted binary trees into a sorted one) use nested recursion. In this case is difficult to guess an induction principle which can be applied. Our novel combinatorial technique combined with lazy induction is able to find the appropriate induction principle, by guessing a suitable witness term.

During the process of synthesis we use the "cascading" principle. This is the case when the proof of the initial statement fails because auxiliary functions are missing. By analysis of the failing proof we generate conjectures which lead to the synthesis of auxiliary functions (e. g. *Insert*, *Merge*).

By theory exploration in *Theorema* we produce 3 axioms, 11 definitions, and more than 200 properties. In the *Coq* system we produce 26 definitions and 50 lemmas. These theories are useful for the further study of properties and algorithms on binary trees. Although produced "manually", these theories can be of interest for the study of possible automation of the theory exploration process.

Related Work. Algorithm synthesis and program generation is currently a challenging problem for programming and verification communities.[2] Note however that our work does not address *program generation*, which consists in transforming the description of an algorithm given in a certain formalism, into a program expressed in a certain language.

Automation of *Theory Exploration* is an active area of research – see e. g. (Colton, 2012), however in our research we did not focus on the mechanization of this process. Rather, we apply the basic principles described in (Buchberger, 2000) in order to perform top–down in parallel with bottom–up exploration, by adding the notions and properties which are necessary for our proofs.

[2] http://research.microsoft.com/en-us/um/people/sumitg/pubs/synthesis.html

Induction Reasoning is used successfully for algorithm synthesis – see e. g. (Bundy et al., 2006; Johansson et al., 2011; McCasland and Bundy, 2006). Powerful techniques (for instance *rippling*) have been developed for overcoming a basic drawback of explicit induction: it may be that the desired algorithm cannot be constructed using the apriori given induction. Our research complements this efforts by introducing a combinatorial technique for the generation of the witness terms, as well as a lazy induction method based on Noetherian ordering, which is able to discover new explicit induction schemes.

We perform *Algorithm Synthesis* by following the classical proof–based synthesis approach as presented in e. g. (Bundy et al., 2006). The work of (Gulwani, 2010) is a comprehensive overview of the most common approaches used to tackle the synthesis problem. In (Basin et al., 2004) we find a comparison between three synthesis methods: constructive/deductive synthesis, schema-based synthesis and inductive synthesis. We do not focus on the theoretical aspects of algorithm synthesis, but we follow an experimental approach in which we can find efficient proof methods. Concerning the sorting of binary trees, in classical approaches – see e. g. (Knuth, 1998) the problem of sorting them directly is not investigated, and we did not find other descriptions of such algorithms.

Section 7 discusses related work in more detail.

Our previous work on lists (Dramnesc and Jebelean, 2011, 2012c,b, 2015b) is extended by the research presented here. We introduce new induction principles, proof strategies and inference rules. Some of them are based on the properties of binary trees, but most are general, and in fact the combinatorial technique for generating witnesses can also be used for lists, where it can generate novel versions of `Merge` on lists (merging two sorted lists into a sorted one).

The "cascading" principle (Buchberger and Craciun, 2004), was also used for lists (Dramnesc and Jebelean, 2015b). This paper describes in more detail the "cascading" principle and the techniques.

A similar proof–based approach has been experimented in (Dramnesc and Jebelean, 2012a) and (Dramnesc and Jebelean, 2015a) in order to synthesize algorithms operating on *monotone lists*, however in that case the nature of the problem imposes a focus on equality–based rewriting methods, rather than relational ones.

Our research on binary trees has been partially presented in some conferences and workshops (Dramnesc et al., 2015c, 2016b, 2015a, 2016a). The present paper constitutes a synthesis of all our results on binary trees but it adds new ideas and improves the presentation of our work as follows.

Based on the case studies on the two different data structures (lists and binary trees), we give a more comprehensive classification and a generalization of the proof techniques and we illustrate more clearly their application for the experiments on binary trees. For instance the combinatorial technique first presented in (Dramnesc et al., 2015a) is now refined and discussed in more detail, together with a comprehensive illustration.

We extend the experiments in *Coq* to certify all the synthesized sorting algorithms and discuss in more detail the differences between the experiments performed in *Theorema* and in *Coq*.

Structure of the paper. Section 2 presents the theoretical framework and the basic notions, Section 3 presents the proof techniques which we use, Section 4 presents the exploration of the theory of binary trees, Section 5 presents the synthesis experiments, Section 6 presents the work done in the *Coq* system, Section 7 presents the differences/similarities with related work and Section 8 makes the final remarks.

2. Theoretical Framework

This section presents the context of binary trees and the notations, introduces the synthesis problem and describes the main synthesis method: the construction of the synthesis conjecture, its proof by induction, and the extraction of the algorithm from the proof.

The results presented in this section have been partially published in (Dramnesc et al., 2016b, 2015a), see also (Dramnesc et al., 2015b).

2.1. General Conventions

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 $\underset{\overline{x}}{\forall}\Phi[\overline{x}]$ by using *substitutions* that map subsets of free variables from $\Phi[\overline{x}]$ 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 $\Phi[\overline{x}]$. We denote the application of a substitution σ to a formula Φ by $\Phi\sigma$ and say that $\Phi\sigma$ is an *instance* of Φ. An *identity* substitution, generically denoted by σ_{id}, maps every 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 vectors of terms.

All object statements are logical formulas, but for a better understanding we label them as **Definition** and **Axiom** (with the obvious meaning), as well as **Property** (a statement which has to be proven from other statements), and **Conjecture** (a statement allowing to extract an algorithm from its possible proof).

2.2. Sorting of Binary Trees

We consider *binary trees* over *elements* from a domain with a total ordering. The two types however are not explicitly present in the formulas, but only implicit by notation and 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.

From the names of variables present in the original formulas, the provers generates meta–variables (denoted usually by starred symbols — e.g., T^*, T_1^*, Z^*) and Skolem constants (e.g., X_0, X_1, a_0).

The ordering between tree elements is denoted by the usual $\leq$, and the ordering between a tree and an element is 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).

We use two constructors for binary trees, namely: 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.

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. 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) $L \preceq a \preceq R$, and ii) L and R are sorted trees.

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

The formal definitions of these predicates and functions are given in Section 4.

2.3. Synthesis Problem and Method

As stated in the introduction, the *specification* of the target function F consists of two predicates: the input condition $I[X]$ and the output condition $O[X, T]$, and the correctness property for F is $\underset{X}{\forall}(I[X] \Rightarrow O[X, F[X]])$. The synthesis problem is expressed by the conjecture: $\underset{X}{\forall}\underset{T}{\exists}(I[X] \Rightarrow O[X, T])$. The proof-based synthesis consists in proving this conjecture in a constructive way and then extracting the algorithm for the computation of F from this proof.

In the case of sorting, the input condition specifies the type of the input, therefore it is missing since the type is implicit. The output condition $O[X, T]$ is $X \approx T \wedge IsSorted[T]$ thus the synthesis conjecture becomes:

Conjecture 1. $\underset{X}{\forall}\underset{T}{\exists}(X \approx T \wedge IsSorted[T])$

This conjecture can be proved in several ways. Each constructive proof is different depending on the applied induction principle and the content of the knowledge base (the set of assumptions provided to the prover). Hence, different algorithms are extracted from different proofs.

Synthesis scenarios. The simple scenario is when the proof succeeds, because the properties of the auxiliary functions which are necessary for the implementation of the algorithm are already present in the knowledge base. The auxiliary algorithms used for tree sorting are $Insert[a, A]$ and $Merge[A, B]$. Their necessary properties are given in Section 4.

More complex is the scenario where the auxiliary functions are not present in the knowledge base. In this case the prover fails and on the failing proof situation we apply ***cascading***: we create a conjecture which would make the proof succeed, and it also expresses the synthesis problem for the missing auxiliary function. In this scenario, the functions *Insert* and *Merge* are synthesized in separate proofs, and the main proof is replayed with a larger knowledge base which contains their properties.

2.4. Induction Principles and Algorithm Extraction

The illustration of the induction principles and algorithm extraction in this subsection is similar to the one from (Dramnesc and Jebelean, 2011), but the induction principles are adapted for trees and the extracted algorithms are more complex.

The following induction principles are direct *term-based* instances of the Noetherian induction principle (Stratulat, 2012) and can be represented using *induction schemas*. Consider the domain of binary trees with a well-founded ordering $<_t$ and denote by $\ll_t$ the multiset extension (Baader and Nipkow, 1998) of $<_t$ as a well-founded ordering over vectors of binary trees. An induction schema to be applied to a predicate $\underset{\overline{x}}{\forall}P[\overline{x}]$ defined over a vector of tree 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 trees. An induction schema may attach to an induction conclusion

$P[\bar{t}]$, as *induction hypotheses*, any instance $P[\bar{t'}]$ of $\underset{\bar{x}}{\forall} P[\bar{x}]$ as long as $\bar{t'} \ll_t \bar{t}$. The induction conclusions without (resp., with) attached induction hypotheses are *base* (resp., *step*) cases of the induction schema.

In the current presentation we will use the *multiset of elements* as the measure of binary trees. Checking strict ordering $E <_t E'$ between two expressions E, E' representing trees reduces to check strict inclusion between the multisets of symbols (constants and variables except ε) occurring in the expressions. This is because the expressions representing trees contain only functions which preserve the number of elements i.e., the elements of the returned tree, on the one hand, and from the arguments, on the other hand, build equivalent multisets (*Concat, Insert, Merge*).

In our experiments, we use the following induction principles for proving P as unary predicates over binary trees.

$$\textbf{Induction-1: } \Big(P[\varepsilon] \wedge \underset{n,L,R}{\forall}((P[L] \wedge P[R]) \Longrightarrow P[\langle L, n, R\rangle])\Big) \Longrightarrow \underset{X}{\forall} P[X]$$

This is the standard structural induction based on the definition of trees.

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 order to synthesize the sorting algorithm as a function $F[X]$, we consider the output condition $O[X, T] : (X \approx T \wedge \mathit{IsSorted}[T])$. **Induction-1** can be applied to prove the synthesis conjecture $\underset{X}{\forall}\underset{T}{\exists} O[X, T]$ by taking $P[X]$ as $\underset{T}{\exists} O[X, T]$.

The proof is structured as follows:

Base case: We prove $\underset{T}{\exists} O[\varepsilon, T]$. If the proof succeeds to find a ground witness $\mathfrak{I}_1$ such that $O[\varepsilon, \mathfrak{I}_1]$, then we know that $F[\varepsilon] = \mathfrak{I}_1$.

Step case: For arbitrary but fixed n, L_0 and R_0 (new constants), we prove $\underset{T}{\exists} O[\langle L_0, n, R_0\rangle, T]$. We assume as induction hypotheses $\underset{T}{\exists} O[L_0, T]$ and $\underset{T}{\exists} O[R_0, T]$, which are Skolemized by introducing two new constants T_1 and T_2 for each existential T. The existential 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 $T^* = \mathfrak{I}_2[n, L_0, R_0, T_1, T_2]$ (term depending on n, L_0, R_0, T_1 and T_2), then we know that $F[\langle L, n, R\rangle] = \mathfrak{I}_2[n, L, R, F[L], F[R]]$.[3]

The extracted algorithm from the proof is expressed as:

$$\underset{n,L,R}{\forall}\left(\begin{array}{c} F[\varepsilon] = \mathfrak{I}_1 \\ F[\langle L, n, R\rangle] = \mathfrak{I}_2[n, L, R, F[L], F[R]] \end{array}\right)$$

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 (Bundy et al., 2006).

[3] T_1 and T_2 are replaced by $F[L]$ and $F[R]$, respectively.

For the following induction principles, the proof and the algorithm extraction are similar to **Induction-1**, therefore we give only the structure of the extracted algorithm for each induction principle.

Induction-2:

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

This induction principle was suggested by the experiments, according to the induction schema discovery explained in Section 3.4, and illustrated at the end of that section.

The extracted algorithm is:

$$\underset{n,L,R}{\forall}\begin{pmatrix} F[\varepsilon] = \mathfrak{I}_1 \\ F[\langle L,n,\varepsilon\rangle] = \mathfrak{I}_3[n,L,F[L]] \\ F[\langle L,n,R\rangle] = \mathfrak{I}_5[n,L,R,F[\langle L,n,\varepsilon\rangle],F[R]] \end{pmatrix}$$

Induction-3:

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

This is an expression of Induction-1 which makes explicit the cases of empty subtrees, also discovered experimentally using the induction schema discovery. L and R are assumed to be non-empty trees. In order to encode this conveniently during the proof, they are replaced by $\langle A,a,B\rangle$ and $\langle C,b,D\rangle$, respectively.

The extracted algorithm is:

$$\underset{n,a,b,A,B,C,D}{\forall}\begin{pmatrix} F[\varepsilon] = \mathfrak{I}_1 \\ F[\langle\varepsilon,n,\varepsilon\rangle] = \mathfrak{I}_2[n] \\ F[\langle\langle A,a,B\rangle,n,\varepsilon\rangle] = \mathfrak{I}_3[n,A,a,B,F[\langle A,a,B\rangle]] \\ F[\langle\varepsilon,n,\langle C,b,D\rangle\rangle] = \mathfrak{I}_4[n,C,b,D,F[\langle C,b,D\rangle]] \\ F[\langle\langle A,a,B\rangle,n,\langle C,b,D\rangle\rangle] = \mathfrak{I}_5[n,a,b,A,B,C,D, \\ F[\langle A,a,B\rangle],F[\langle C,b,D\rangle]] \end{pmatrix}$$

Other induction principles are also used in the experiments, being discovered automatically by a combinatorial technique – see Subsection 3.3.

3. Special Proof Techniques

One important result of this research is the revelation of certain proof techniques (special inference rules and strategies) which are useful for the development of the proofs performed during the experiments. These combine a few already known techniques from natural–style proving with numerous new ones. We present in this section first the techniques in their general form, and after that their specific realisation as inference rules and proof strategies.

As expected, some of the techniques employed for the domain of lists (Dramnesc and Jebelean, 2015b) also extend here.

The results presented in this section have been partially published in (Dramnesc et al., 2016b, 2015a), see also (Dramnesc et al., 2015b).

3.1. Natural–Style Techniques

Following the spirit of the *Theorema system*, we aim to generate proofs in *natural style* – that is, in a style similar to proofs developed by humans. Note that by natural style we do not mean *natural deduction*, as described e. g. in (Pelletier, 2000). What we mean are just some general requirements for the rules for presentation of mathematical statements and proofs, as well as for logical inferences, such that they are similar to the ones used by human scientists - see (Buchberger et al., 2016). We here briefly survey the techniques used in our experiments, which are not really novel, as they have been used in our previous experiments and also in previous work in the *Theorema system* and possibly in other provers. However we describe them here in order to improve the readability of our approach. These techniques are not specific to algorithm synthesis or to the domain of binary trees, but can be used in any context.

In general, the proof proceeds by transforming at each step a *proof situation* (consisting of a set of *assumptions* and one *goal*) by an *inference rule*. The inference steps can be classified as *forward* steps (by which one or more assumptions produces one or more assumptions) and *backward* steps (by which the goal is transformed and possibly one or more goals are produced). Sometimes several proof situations are produced, which are either connected by an *and* node (all proofs must succeed) or by an *or* node (at least one proof must succeed). The order in which the inference steps are applied is decided by *proof strategies*. The proof succeeds when the goal coincides with (or is an instance of) an assumption. (Proof success is also accepted in certain situations when the goal can be used as a condition in a clause of the synthesized algorithm – see below).

Induction using the structural definition of the domain. The use of induction is described in Subsection 2.4, and the particular induction principle used by a certain proof is chosen by the user. In some special situations, the prover refines the induction principle (see Subsection 3.4).

The induction step is combined with the inferences related to quantifiers – this is a proof strategy typical for natural–style proving (see **Example 4** below).

Skolemization of universal goals. This consists in replacing the universally quantified variables of the goal by constants (the so–called "arbitrary but fixed" constants from natural–style proving), and is typically used at the beginning of the induction step.

Skolemization of existential assumptions. This consists in replacing the existentially quantified variables of an assumption by constants (the so–called "take such a" constant from natural–style proving), and it is used *only* during the induction step. Namely, the transformation is applied only to the existential variables stating the existence of the sorted trees in the left-hand side of the implication occurring in the induction principle. (These constants will be transformed in recursive calls of the sorting function in the synthesized algorithm.) In other situations, Skolemization of existential assumptions would lead to non-constructive proofs (the generated constants will not correspond to any algorithm for computing their values).

Meta–variables for existential goals. This consists in replacing the existentially quantified variables of the goal with meta–variables. Meta–variables stand for terms which have to be found, and they are essentially equivalent to the variables of resolution calculus. In our examples the goal may contain only such variables. Note however that we use meta–variables only for existential goals, and not for universal assumptions.

The solution for a meta–variable may not depend on the Skolem constants which are introduced in the proof *after* the creation of the corresponding meta–variable. This is checked using a partial order between the meta–variables and the Skolem constants – see e.g. (Konev and Jebelean, 2001).

Goal reduction: SLD deduction. The goal is always an atom or a conjunction of ground atoms, which has to be reduced until it becomes an instance of an assumption. Except for the special inferences described later, the goal is reduced using the *head* of assumptions (the atom on the right-hand side of the [universal] implication, or the assumption itself if it is atomic). An atom from the goal is reduced with priority if it unifies with an assumption. If an atom from the goal unifies with the right-hand side of an assumed implication, then the instantiated left-hand side (sometimes new meta–variables may be necessary) replaces the respective atom in the goal. If this happens for several assumptions, then an *or* node is produced, with a branch for each possibility. This is essentially equivalent to the *SLD–resolution style* (Selective Linear Definite clause resolution) of proving (Kowalski and Kuehner, 1971; Mordechai, 2004), which is used in Prolog. The use of this kind of reasoning is possible because we use first-order predicate logic, and most of the formulas are essentially equivalent to Horn clauses. Moreover, the goal is always a conjunction of atoms.

For improved efficiency, we do not apply SLD exactly in the same manner, but we refine it using the rule *Eliminate-Ground-Formulas-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.

However, applying the *SLD–resolution style* easily leads to an explosion of the search space, because one has to generate branches for all possible matchings, like in Prolog. Moreover, certain properties (like e.g. the transitivity and reflexivity of equivalence relations) easily create infinite loops. In practice, this proving mechanism is often not able to generate complex proofs because the exhaustion of time or space resources.

In order to make the proof process more efficient and to avoid the search space explosion, we use certain special inference rules and strategies. These increase the efficiency of proving in a very significant way, and in fact one of the main results of our experiments is the discovery and the demonstration of such specific inference rules and strategies.

3.2. Novel Proof Techniques

Although these new proof techniques have been revealed during experiments on specific domains (previously – lists and currently – binary trees), all these techniques are quite general, and can be applied on arbitrary domains. Some of them, however, will be applicable only if the domain has certain type of properties, like a special equivalence relation which is preserved by certain functions and predicates, or certain ordering relations, as we discuss below.

Restricted use of universal assumptions. A universal assumption is used in a *backward* step only if the conclusion of the implication unifies with a conjunct of the goal: in this case, the instantiated premises of the implication will replace the respective conjunct. A universal assumption is used in a *forward* step only together with a local assumption created during the proof (which is typically ground).

Priority of local assumptions in forward inferences. We classify assumptions in *global* and *local*. Global assumptions are the ones from the initial proof situation, and they generally hold for the domain. (Typically they are universally quantified implications.) Local assumptions are generated during the proof (like e.g. the premises of a goal implication), and they hold only as temporary assumptions during the proof. Priority of local assumptions means that, in backward inferences, we first try to unify the goal against local assumptions, and that, in forward inferences, at least one of the used assumption is a local one. This strategy is essentially equivalent with the "set of support" strategy in clausal resolution.

No existential Skolemization outside induction hypotheses. As mentioned before, Skolemization of the existential assumptions is applied only when the generated constants correspond to recursive calls of the synthesized algorithm.

Decomposition into microatoms. By *microatom* we understand an atom whose arguments are just constants (it does not contain function symbols). By the properties of the domain, sometimes an atom can be decomposed into a set of microatoms. This is done whenever possible both in the assumptions and in the goal. The transformation is performed differently depending on where the atom to be transformed occurs in the proof situation. If the atom is an assumption, then the rule generates as many microatoms as possible, as individual assumptions. If the atom is [part of] a goal, then the rule generates as few microatoms as possible, and they become conjuncts in the goal. In this way, some of the goal conjuncts will match some of the assumptions, and the goal is simplified. Moreover, some of these microatoms will become conditional assumptions in the synthesized algorithm (see *Precondition from goals* below).

Priority of strong constants in goal. By *strong* constants, we understand the constants which occur in more assumptions (they have more properties). The technique consists in replacing in the goal the constants by stronger ones whenever possible. In the particular case of the algorithm synthesis proofs, such strong constants are usually Skolem constants generated by existentially quantified assumptions, which are induction hypotheses. Therefore, they will correspond to recursive calls.

Preconditions from goals. We call a goal *simple* if it cannot be reduced using the current inference rules and assumptions, and it does not contain binary trees as arguments. (In the current experiments we make an exception however for *RgM* and *LfM* applied to constants.) This technique consists in considering as successful a proof situation whose goal is simple, while the simple goal is used as a precondition for the current witness in the synthesized algorithm. On alternative branches further conditional clauses of the current algorithm will be synthesized.

This technique allows the generation of conditional clauses in algorithms (essentially `if-then-else` structures).

Techniques Using Equivalence, Ordering, and Noetherian Ordering

The main idea of these techniques is to lift properties of specific predicates and functions at inference level, that is to create inference rules based on these properties.

The fact that the relation $\approx$ (having the same elements) is an equivalence gives term replacement rules in atoms using $\approx$ and $\preceq$, because these two predicates are equivalence preserving. Also *Insertion, Merge, Concat* are equivalence preserving, thus equivalent subterms may be replaced without altering the equivalence property.

Certain relationships between $\approx, \leq, \preceq$, *IsSorted, Insertion, Merge, Concat, LfM, RgM* are used for combinatorial generation of witnesses and of their preconditions, and for decomposition into microatoms.

More importantly, the equivalence $\approx$ has as normal form the multiset of the elements of the respective tree. This normal form is not used explicitly in our experiments, but gives a basis for some of the proof techniques. Namely, the equality of the multisets expresses $\approx$, and this extends naturally to a meta–equivalence $\approx_t$ between what we call $\approx$*–preserving ground terms*: the terms which use only the tree constructors and the $\approx$–preserving function symbols (*Sort, Insertion, Merge, Concat*). For this meta–equivalence, the normal form is the multiset of constants occurring in the respective term, thus the equivalence of terms is very easy to check syntactically. This aspect is used in the implementation of the replacement rules mentioned above.

Moreover, the strict inclusion of the multiset of elements of the trees constitutes a Noetherian relation over this domain, we denote it by $\ll$, which extends in the same way to the Noetherian meta–ordering $\ll_t$ over the same set of $\approx$–preserving ground terms. These two Noetherian orderings play a very important role in the proof–driven refinement of the induction principle – see below.

Note that $\approx$ and $\ll$ are tightly related: they both come from the partial ordering induced by the inclusion of multisets of elements. The equivalence $\approx$ is induced by multiset equality, the strict ordering $\ll$ is induced by strict inclusion of multisets, and both relations extend naturally to meta–relations on the same set of (important) ground terms. Exactly the same happens for the domain of lists.

For easier understanding we presented these techniques with respect to the specific predicates and functions over binary trees, but clearly they are usable in the same way for lists, and in fact for any domain which exhibits predicates and functions with similar properties.

3.3. A Combinatorial Technique for Finding Witness Terms

We detail here the combinatorial technique which we use in order to synthesize the function for merging two sorted binary trees into a sorted binary tree. 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 this combinatorial technique and the Noetherian induction on the relations $\ll$ and $\ll_t$ presented in the previous subsection.

This technique is applied when the proof needs a sorted witness which is equivalent with a certain ground term T, and there is no other inference rule to apply. It consists in generating all possible $\approx_t$–equivalent terms – by taking all $\approx$–preserving ground terms which have the same multiset of ground constants as T. Based on the Noetherian induction principle, in these terms we also accept the function symbol to be synthesized, as long as it heads a term whose multiset of constants is strictly included in the multiset of constants of T. Such terms lead to new explicit induction principles (and new recursion structures) which have not been apriori indicated to the prover – as explained in the next subsection.

Note that this technique can be used in proofs on any domain which has an appropriate Noetherian ordering and equivalence relation, for instance on the domain of lists.

The concrete application of this principle is illustrated in detail in Subsection 5.4.

Concerning the time complexity of this approach, note that the number of possible terms is limited by the (relatively small) list of constants occurring in the term T. The number of possible terms is exponential, because all permutations of this list have to be considered, however in practice this does not increase significantly the running time, because it is quite small compared to the proving time when many proof branches have to be investigated. In our experiments this term generation and selection process was under few seconds.

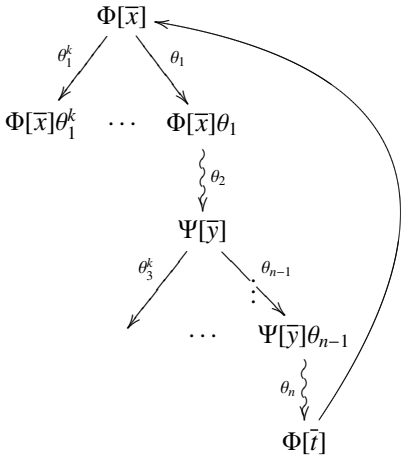
3.4. Refining Induction by Lazy Reasoning

As shown e.g. in (Stratulat, 2012), 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 reiterate 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.

In general, when we want to prove a formula $\forall_{\overline{x}} \Phi[\overline{x}]$ 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 $\Phi[\overline{x}]$ are encountered. An instance $\Phi[\overline{t}]$ can be used as induction hypothesis for the induction case $\Phi[\overline{x}]\theta$ if $\overline{t}$ is smaller than $\overline{x}\theta$, with respect to the Noetherian ordering $\ll_t$ on terms.

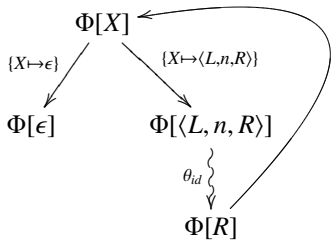
The substitution θ, called *cumulative* substitution, is built from the proof. To illustrate its computation, we represent the proof derivation as a tree for which the root node is labeled by $\Phi[\overline{x}]$. Two kinds of non-root nodes are distinguished: instantiation nodes and deductive nodes. 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 $\Phi[X]$, a covering set of instance formulas is $\{\Phi\{X \mapsto \epsilon\}, \Phi\{X \mapsto \langle L, n, R\rangle\}\}$, 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 $\Phi[\overline{x}]$, to the node labeled by the induction hypothesis, in our case $\Phi[\overline{t}]$. This scenario can be illustrated as below:



In our scenario, $\Phi[\overline{t}]$ is an instance of $\Phi[\overline{x}]$. In addition, it can be used as an induction hypothesis if $\overline{t}$ is smaller than $\overline{x}\theta_1\theta_2\cdots\theta_{n-1}\theta_n$.

Example 2. *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 $\Phi[X]$*, where X is a binary tree:*

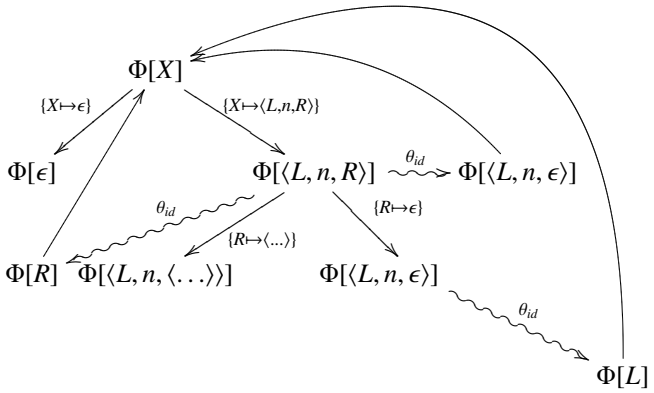


Here, θ_{id} is the identity substitution $\{L \mapsto L; n \mapsto n; R \mapsto R\}$. $\Phi[R]$ can be used as induction hypothesis in the proof of the case $\Phi[\langle L, n, R\rangle]$ because R has a number of elements smaller than $\langle L, n, R\rangle$.

The corresponding explicit induction principle is:

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

Example 3. *More specific induction schemas can also be generated by lazy induction, as shown in the following scenario:*



The corresponding explicit induction principle is:

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

*Note that this is **Induction-2** that has been developed during our experiments, during the proof of **Conjecture 1** presented in Subsection 5.3.1.*

3.5. Realisation of the Techniques

The proof techniques presented above are realised as *Inference Rules* (**IR-1**, **IR-2**, . . .) and *Proof Strategies* (**S-1**, **S-2**, . . .) in the prover for the special domain of binary trees.

3.5.1. Inference Rules

IR-1: *Generate Microatoms.* In the context of binary trees, we allow the functions *RgM* and *LfM* in microatoms.

For instance, $IsSorted[\langle T_1, n, T_2\rangle]$ is transformed into $(IsSorted[T_1] \wedge IsSorted[T_2] \wedge RgM[T_1] \leq n \wedge n \leq LfM[T_2])$. Similarly, $x \preceq \langle A, b, C\rangle$ is transformed into $x \preceq A \wedge x \preceq b \wedge x \preceq C$.

IR-2: *Eliminate-Ground-Formulas-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.

Example: One of the assumptions is $IsSorted[T_2]$ and the goal is $\langle T_1, n, T_2\rangle \approx T^* \wedge IsSorted[T_1] \wedge IsSorted[T_2]$. The goal is transformed into: $\langle T_1, n, T_2\rangle \approx T^* \wedge IsSorted[T_1]$. This is a simple refinement of the Prolog style mechanism, which increases the proof efficiency.

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.

Example: Among the assumptions are: $\langle L_1, n_1, R_1\rangle \approx T_1$ and $\langle L_2, n_2, R_2\rangle \approx T_2$ and the goal is: $\langle\langle L_1, n_1, R_1\rangle, n, \langle L_2, n_2, R_2\rangle\rangle \approx T^*$, then the new goal becomes: $\langle T_1, n, T_2\rangle \approx T^*$.

These replacements are performed according to certain heuristics. In this example T_1, T_2 are stronger constants (see Subsection 3.2).

This rule has several refinements which we present below.

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

Example: The assumption is: $\langle \varepsilon, n, L\rangle \approx T_1$ and the goal is: $\langle \varepsilon, n, \langle L, m, R\rangle\rangle \approx T^* \wedge IsSorted[T^*]$, then the new goal becomes: $\langle T_1, m, R\rangle \approx T^* \wedge IsSorted[T^*]$.

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 constructors.

Example: The assumption is: $Concat[L, S] \approx T_1$ and the goal is: $Concat[\langle L, n, R\rangle, S] \approx T^* \wedge IsSorted[T^*]$, then the new goal becomes: $\langle T_1, n, R\rangle \approx T^* \wedge IsSorted[T^*]$.

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.

Example: The assumption is: $\langle \varepsilon, n, L\rangle \approx T_1$ and the goal is: $RgM[T_1] \leq m \wedge m \leq LfM[R]$, then the first conjunct of the goal changes and the new goal becomes: $n \leq m \wedge L \preceq m \wedge m \leq LfM[R]$.

Note that, in all these transformations, it is not guaranteed that the new goal is provable, therefore they are applied by generating proof alternatives.

The next rule is applied only after all the transformation rules presented above have been applied. This is because it generates many branches in the proof tree.

IR-4: *Generate permutations.* This rule implements the combinatorial technique presented in Subsection 3.3. When the goal is of the form $Expression \approx T^* \wedge IsSorted[T^*]$, the prover generates permutations of the list of non-empty arguments present in *Expression* and for each permutation it generates witnesses as a tree expressions containing these elements. These expressions can contain: the constructor $\langle \ldots \rangle$, and the functions *Insert*, *Concat*, *Merge*, and *Sort*.

For instance, if *Expression* is ε then only the tree ε is generated. If *Expression* is $\langle L, x, \varepsilon\rangle$, then the generated tree expressions are: $\langle L, x, \varepsilon\rangle$, $\langle \varepsilon, x, L\rangle$, and $Insert[n, L]$.

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 $IsSorted[L]$ 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 Section 5.4) 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.

The prover tries an alternative for each generated witness as a solution for the T^* meta–variable, and an additional alternative if the proof does not succeed – see strategy **S-3.**

The use of this rule is optional, depending on the preferences set by the user. Moreover, the user can specify which functions are to be used for generating the expressions. In particular, one may use *Insert*, *Merge* and *Sort*[4] only in certain situations. For instance, *Merge* cannot be used if it is not yet defined, except in the synthesis of *Merge* itself, but then the prover checks that the arguments of its usage are smaller (w.r.t. the multiset of symbols) than the arguments of the expression which is synthesized on the current branch of the induction – see formula (23).

The advantage of applying this rule is that one obtains various solutions for some branches of the algorithm, which may lead to more efficient computation when the input has certain specific properties. More details about generating the permutations are given in Subsection 3.3.

IR-5: *Simple-Goal-Conditional-Assumption.* This applies when the goal is ground, no further simplification of it is possible, and the goal does not contain tree constants except inside the functions *RgM*, and *LfM*. This goal becomes a conditional assumption representing the condition attached to the corresponding branch of the synthesized algorithm, and the current branch is considered successful (according to strategy **S-3**). The reason for the selection of such formulas as conditional assumptions is that they can be easily evaluated (an expression which does not contain tree expressions is evaluated in constant time, and the functions *RgM* and *LfM*, are evaluated in linear time).

One example of such a conditional assumption is $m \leq n$, and a more complex one is: $RgM[\langle A_2, z_2, B_2\rangle] \leq n \wedge m \leq LfM[\langle A_2, z_2, B_2\rangle]$.

3.5.2. Specific 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), and it is used for goals in the induction steps.

The following example illustrates the use of the quantifier-related techniques during the induction step.

Example 4. *Consider the proof in Subsection 5.1 at page 23, namely the step* Induction case 1. *This corresponds to the second conjunct in the left-hand side of* **Induction-3** *(Subsection 2.4 at page 6), namely* $\underset{n,L}{\forall}(P[L] \Longrightarrow P[\langle L, n, \varepsilon\rangle])$, *where* $P[X]$ *is* $\underset{T}{\exists}(X \approx T \wedge IsSorted[T])$. *First the universal quantifier is eliminated by transforming the quantified variables n and L into (so–called "arbitary but fixed") constants n and* L_0, *then (by the so called "deduction rule"), the goal generates the assumption:*

$$\underset{T}{\exists}(L_0 \approx T \wedge IsSorted[T])$$

and the new goal:

$$\underset{T}{\exists}(\langle L_0, n, \varepsilon\rangle \approx T \wedge IsSorted[T]).$$

By Skolemization of the existential variable the assumption becomes:

$$L_0 \approx T_1, \;\; IsSorted[T_1]$$

[4]Note that we use F_1, F_2, etc. for different versions of *Sort*

and by introducing a meta–variable for the unknown necessary witness the goal becomes:

$$\langle L_0, n, \varepsilon \rangle \approx T^* \wedge IsSorted[T^*]$$

The proof succeeds if an appropriate solution (term) for the meta–variable $T^*$is found, typically this is the witness which will be used in the synthesized algorithm.

S-2: *Priority-of-Local-Assumptions.* This strategy implements the corresponding technique described in Subsection 3.2.

S-3: *Preconditions-from-goals.* This strategy implements the corresponding technique described in Subsection 3.2. The prover generates several proof branches using rule **IR-4**, follows each branch one by one and produces a set of conditional witnesses which becomes a multiple branch in the synthesized algorithm. The final proof is successful if the disjunction of all conditions is true – this means that the algorithm covers all possible cases. *Example:* on one branch, one obtains the condition $m \leq n$ and, on another branch, the condition $n \leq m$.

4. Theory Exploration

By *theory exploration* (see (Buchberger, 2004)), we understand the process of creating the theory of the investigated domain, during the algorithm synthesis experiments. The exploration is an alternation of bottom–up and top–down processes. In the bottom–up way, we introduce the necessary statements, starting from axioms, definitions, and obvious properties (which are also proven automatically). In the top–down way, we introduce auxiliary notions and properties when we detect that they can be useful for the proofs which are performed.

This process of theory exploration is supported by the *Theorema* system in the following way: the formulas can be expressed in natural style, including two-dimensional notations; the knowledge can be labeled and organized hierarchically in a natural way; the statements can be proven automatically, and when this fails the natural–style proofs help to identify the cause (missing assumptions, wrong formulations, etc.) We did not focus in our research on the automation of theory exploration, which is performed for instance in (Johansson et al., 2011; McCasland and Bundy, 2006), however the theory created and the experience from our experiments may be useful for further investigation of possible heuristics and algorithms for such automation. (A similar but reduced bottom-up process is performed in the *Coq* system in order to certify the synthesized algorithms – see Section 6).

This section describes part of the formal axioms, definitions and properties used in the theory of binary trees. The full theory contains 3 axioms, 11 definitions, and more than 200 properties. The theory also includes the synthesis conjectures (which after proof become theorems), as well as definitions of algorithms which are synthesized for sorting and for the auxiliary functions, but these are presented in the Section 5.

The results presented in this section have been partially published in (Dramnesc et al., 2015c).

4.1. Basic Notions

4.1.1. The Relation '≤' Between Elements

Property 1. $\underset{a,b,c}{\forall}\Big((a \leq b \wedge b \leq c) \Longrightarrow a \leq c\Big)$

4.1.2. The Relation '$\preceq$' Between an Element and a Tree

Definition 5. $\underset{n,L}{\forall}\Big(\underset{Member[m,L]}{\forall}\Big(n \preceq m\Big) \Longleftrightarrow n \preceq L\Big)$

Property 2. $\underset{m}{\forall}\Big(m \preceq \varepsilon \ \wedge \ \varepsilon \preceq m\Big)$

Property 3. $\underset{m,n,L,R}{\forall}\Big((m \preceq L \wedge m \preceq n \wedge m \preceq R) \Longrightarrow m \preceq \langle L, n, R\rangle\Big)$

Property 4. $\underset{a,b,L}{\forall}\Big((a \preceq b \wedge b \preceq L) \Longrightarrow a \preceq L\Big)$

4.1.3. The Relation '$\underline{\ll}$' Between Trees

Definition 6. $\underset{L,R}{\forall}\Big(L \underline{\ll} R \Longleftrightarrow \underset{\substack{Member[m,L]\\Member[n,R]}}{\forall}\Big(m \preceq n\Big)\Big)$

Property 5. $\underset{L}{\forall}\Big(L \underline{\ll} \varepsilon\Big)$

Property 6. $\underset{L}{\forall}\Big(\varepsilon \underline{\ll} L\Big)$

Property 7. $\underset{n,L,R,S}{\forall}\Big((L \underline{\ll} S \ \wedge n \preceq S \ \wedge R \underline{\ll} S) \Longleftrightarrow (\langle L, n, R\rangle \underline{\ll} S)\Big)$

Property 8. $\underset{m,n,L,R,S,T}{\forall}\Big((L \underline{\ll} S \wedge R \underline{\ll} S \ \wedge L \underline{\ll} T \wedge R \underline{\ll} T \wedge m \preceq S \wedge m \preceq T \wedge m \preceq n \wedge L \preceq n \wedge R \preceq n) \Longleftrightarrow (\langle L, m, R\rangle \underline{\ll} \langle S, n, T\rangle)\Big)$

Property 9. $\underset{L,R\neq\epsilon,S}{\forall}\Big((L \underline{\ll} R \wedge R \underline{\ll} S) \Longrightarrow L \underline{\ll} S\Big)$

Property 10. $\underset{n,L,R}{\forall}\Big((L \preceq n \wedge n \preceq R) \Longrightarrow L \underline{\ll} R\Big)$

Property 11. $\underset{m,L\neq\epsilon,R}{\forall}\Big((m \preceq L \wedge L \underline{\ll} R) \Longrightarrow m \preceq R\Big)$

Property 12. $\underset{L,R}{\forall}\Big(L \underline{\ll} R \Longleftrightarrow \underset{Member[m,L]}{\forall}\Big(m \preceq R\Big)\Big)$

Property 13. $\underset{L,R}{\forall}\Big(L \underline{\ll} R \Longleftrightarrow \underset{Member[m,R]}{\forall}\Big(L \preceq m\Big)\Big)$

4.1.4. The Rightmost Element

Definition 7. $\underset{n,m,L,R,S}{\forall}\left(\begin{array}{c} RgM[\langle L, n, \varepsilon\rangle] = n \\ RgM[\langle L, n, \langle R, m, S\rangle\rangle] = RgM[\langle R, m, S\rangle] \end{array}\right)$

4.1.5. The Leftmost Element

Definition 8. $\underset{n,m,L,R,S}{\forall}\left(\begin{array}{c} LfM[\langle \varepsilon, n, R\rangle] = n \\ LfM[\langle\langle L, n, R\rangle, m, S\rangle] = LfM[\langle L, n, R\rangle] \end{array}\right)$

The functions *LfM* and *RgM* do not have a definition for the empty tree, however the following axiom is assumed:

Axiom 1. $\underset{m}{\forall}\Big(RgM[\varepsilon] \preceq m \preceq LfM[\varepsilon]\Big)$.

4.1.6. The Predicate 'IsSorted'

Definition 9.

$$\underset{m,L,R}{\forall}\left(\begin{array}{c} IsSorted[\varepsilon] \\ (IsSorted[L] \wedge IsSorted[R] \wedge RgM[L] \leq m \leq LfM[R]) \Longleftrightarrow IsSorted[\langle L, m, R\rangle] \end{array}\right)$$

This definition is equivalent to:

Definition 10. $\underset{m,L,R}{\forall}\left(\begin{array}{c} IsSorted[\varepsilon] \\ (IsSorted[L] \wedge IsSorted[R] \wedge L \preceq m \preceq R) \Longleftrightarrow IsSorted[\langle L, m, R\rangle] \end{array}\right)$

Property 14. $\underset{z,T}{\forall}\Big(IsSorted[T] \Longrightarrow (T \preceq z \Longleftrightarrow RgM[T] \leq z)\Big)$

Property 15. $\underset{z,T}{\forall}\Big(IsSorted[T] \Longrightarrow (z \preceq T \Longleftrightarrow z \leq LfM[T])\Big)$

Property 16. $\underset{n,L,R}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge L \preceq n \wedge n \preceq R \wedge L \lessdot R) \Longleftrightarrow IsSorted[\langle L, n, R\rangle]\Big)$

4.1.7. The Predicate 'Member'

Definition 11.

$$\underset{a,b,L,R}{\forall}\left(\begin{array}{c} \neg(Member[a, \varepsilon]) \\ Member[a, \langle L, b, R\rangle] \Longleftrightarrow ((a = b) \vee Member[a, L] \ \vee Member[a, R]) \end{array}\right)$$

Property 17. $\underset{a,b,L,R}{\forall}\Big(Member[a, L] \Longrightarrow Member[a, \langle L, b, R\rangle]\Big)$

Property 18. $\underset{a,b,L,R}{\forall}\Big(Member[a, R] \Longrightarrow Member[a, \langle L, b, R\rangle]\Big)$

Property 19. $\underset{a,L,R}{\forall}\Big(Member[a, \langle L, a, R\rangle]\Big)$

4.1.8. The Permutation of a Tree

The definition is not used explicitly in the proofs.
Instead, consider the following properties:

Property 20. $\underset{a,L,R}{\forall}\Big((L \approx R) \Longrightarrow (Member[a, L] \Longleftrightarrow Member[a, R])\Big)$

Property 21. $\underset{n,L,R}{\forall}\Big(\langle L, n, R\rangle \approx \langle R, n, L\rangle\Big)$

Note also that the $\approx$ relation is an equivalence (this is used implicitly by the prover).

4.2. Auxiliary Functions

The following functions are used in the definition of the synthesized sorting algorithms.

4.2.1. The Concatenation of Trees

Definition 12. $\underset{n,L,R,S}{\forall}\left(\begin{array}{c} Concat[\varepsilon, R] = R \\ Concat[\langle L, n, R\rangle, S\,] = \langle L, n, Concat[R, S\,]\rangle \end{array}\right)$

A first simple property which can be proven inductively from **Definition 12** is:

Property 22. $\underset{L}{\forall}\Big(Concat[L, \varepsilon] = L\Big)$

Other properties are:

Property 23. $\underset{L,R,S}{\forall}\Big((L \trianglelefteq R \wedge S \trianglelefteq R) \Longleftrightarrow Concat[L, S\,] \trianglelefteq R\Big)$

Property 24. $\underset{L,R,S}{\forall}\Big((L \trianglelefteq R \wedge L \trianglelefteq S\,) \Longleftrightarrow L \trianglelefteq Concat[R, S\,]\Big)$

Property 25. $\underset{n,L,R}{\forall}\Big((n \preceq L \wedge n \preceq R) \Longleftrightarrow n \preceq Concat[L, R]\Big)$

Property 26. $\underset{n,L,R}{\forall}\Big((L \preceq n \wedge R \preceq n) \Longleftrightarrow Concat[L, R] \preceq n\Big)$

Property 27. $\underset{L,R}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge L \trianglelefteq R) \Longleftrightarrow IsSorted[Concat[L, R]]\Big)$

Property 28. $\underset{L,R}{\forall}\Big(Concat[L, R] \approx Concat[R, L]\Big)$

4.2.2. 'Insert' an Element in a Tree

The definition is (22) – see Subsection 5.2 with the properties:

Property 29. $\underset{n,T}{\forall}\Big(Insert[n, T\,] \approx \langle \varepsilon, n, T\rangle\Big)$

Property 30. $\underset{n,L,R}{\forall}\Big(\langle L, n, R\rangle \approx Insert[n, Concat[L, R]]\Big)$

Property 31. $\underset{n,L,R}{\forall}\Big(\langle L, n, R\rangle \approx Insert[n, Concat[R, L]]\Big)$

Property 32. $\underset{n,T}{\forall}\Big(IsSorted[T\,] \Longrightarrow IsSorted[Insert[n, T\,]]\Big)$

4.2.3. The 'Merge' Operation Between Two Trees

The definition is (23) – see Subsection 5.2.

The following two properties are used to simplify the expressions of some algorithms:

Property 33. $\underset{T}{\forall}\Big(Merge[T, \varepsilon] \approx T\Big)$

Property 34. $\underset{T}{\forall}\Big(Merge[\varepsilon, T\,] \approx T\Big)$

Property 35. $\underset{L,R,S}{\forall}\Big((L \trianglelefteq R \wedge L \trianglelefteq S\,) \Longleftrightarrow L \trianglelefteq Merge[R, S\,]\Big)$

Property 36. $\underset{L,R,S}{\forall}\Big((L \trianglelefteq S \wedge R \trianglelefteq S\,) \Longleftrightarrow Merge[L, R] \trianglelefteq S\Big)$

Property 37. $\forall_{n,L,R}\Big((L \preceq n \wedge R \preceq n) \Longleftrightarrow Merge[L,R] \preceq n\Big)$

Property 38. $\forall_{n,L,R}\Big((n \preceq L \wedge n \preceq R) \Longleftrightarrow n \preceq Merge[L,R]\Big)$

Property 39. $\forall_{n,L,R}\Big(\langle L,n,R\rangle \approx Insert[n,Merge[L,R]]\Big)$

Property 40. $\forall_{n,L,R}\Big(\langle L,n,R\rangle \approx Insert[n,Merge[R,L]]\Big)$

Property 41. $\forall_{L,R}\Big((IsSorted[L] \wedge IsSorted[R]) \Longrightarrow IsSorted[Merge[L,R]]\Big)$

Property 42. $\forall_{L,R}\Big(Merge[L,R] \approx Merge[R,L]\Big)$

Property 43. $\forall_{n,L,R,A,B}((\langle L,n,\varepsilon\rangle \approx A \wedge R \approx B) \Longrightarrow \langle L,n,R\rangle \approx Merge[A,B])$

4.3. Other Useful Properties

4.3.1. Using Multisets

Since the relation $\approx$ is an equivalence, reasoning about it often reduces to reasoning about the multiset of elements of the trees, which furthermore reduces to reasoning about the multiset of symbols occurring in an expression which represents a tree (see Subsection 3.2). This is because most of the functions which operate on trees combine the multisets of elements of their arguments (*Concat, Merge, Insert*, and the constructor $\langle\ldots\rangle$).

As an illustration of this principle, 6 properties have been listed below which correspond to some of the permutations of the tree $\langle L,n,Merge[R,S]\rangle$. These properties can be generated automatically by generating permutations and subterms as described in the Subsection 3.3.

Property 44. $\forall_{n,L,R,S}\Big(\langle L,n,Merge[R,S]\rangle \approx Insert[n,Merge[L,Merge[R,S]]]\Big)$

Property 45. $\forall_{n,L,R,S}\Big(\langle L,n,Merge[R,S]\rangle \approx Insert[n,Merge[Merge[L,R],S]]\Big)$

Property 46. $\forall_{n,L,R,S}\Big(\langle L,n,Merge[R,S]\rangle \approx Insert[n,Concat[L,Merge[R,S]]]\Big)$

Property 47. $\forall_{n,L,R,S}\Big(\langle L,n,Merge[R,S]\rangle \approx Insert[n,Merge[Concat[L,R],S]]\Big)$

Property 48. $\forall_{n,L,R,S}\Big(\langle L,n,Merge[R,S]\rangle \approx \langle L,n,Concat[R,S]\rangle\Big)$

Property 49. $\forall_{n,L,R,S}\Big(\langle L,n,Merge[R,S]\rangle \approx \langle Merge[L,R],n,S\rangle\Big)$

Similarly, one can obtain more than 100 properties corresponding to the permutations of the trees: $\langle L,n,Concat[R,S]\rangle$, $\langle Merge[L,R],n,S\rangle$, and $\langle Concat[L,R],n,S\rangle$. These properties can be extended for more complex trees.

Such automatic generation may contribute to a (partial) mechanization of the theory exploration, however in our experiments we did not pursue this direction of research. This is because it did not appear to be very useful for increasing proving efficiency. In contrast, it appears to be more useful to generate combinatorially the expressions which correspond to the current goal of the proof, instead of generating many universal equivalences and then trying to find the ones which match the current goal.

4.3.2. Decomposition Into Microatoms

During the proof development, it is sometimes useful to replace an atom whose arguments include tree expressions into several atoms whose arguments consist only of variables or constants (microatoms) as explained in 3.2.

Some properties which are necessary for such decompositions are the following:

Property 50. $\underset{n,L,R,S}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge IsSorted[S]) \Longrightarrow IsSorted[Insert[n, Merge[L, Merge[R, S]]]]\Big)$

Property 51. $\underset{n,L,R,S}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge IsSorted[S]) \Longrightarrow IsSorted[Insert[n, Merge[Merge[L, R], S]]]\Big)$

The proofs of **Properties 50** and **51** come directly from **Properties 32**, **41**.

Property 52. $\underset{n,L,R,S}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge IsSorted[S] \wedge L \underline{\ll} R \wedge L \underline{\ll} S) \Longrightarrow IsSorted[Insert[n, Concat[L, Merge[R, S]]]]\Big)$

Property 53. $\underset{n,L,R,S}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge IsSorted[S] \wedge L \underline{\ll} R) \Longrightarrow IsSorted[Insert[n, Merge[Concat[L, R], S]]]\Big)$

The proof of **Property 53** comes directly from **Properties 24**, **27**, **32**, and **41**.

Property 54. $\underset{n,L,R,S}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge IsSorted[S] \wedge L \preceq n \wedge n \preceq R \wedge n \preceq S \wedge R \underline{\ll} S) \Longrightarrow IsSorted[\langle L, n, Concat[R, S]\rangle]\Big)$

Property 54 is easily proved using **Properties 16**, **24**, **25**, **27**, and **9**.

Property 55. $\underset{n,L,R,S}{\forall}\Big((IsSorted[L] \wedge IsSorted[R] \wedge IsSorted[S] \wedge L \preceq n \wedge R \preceq n \wedge n \preceq S) \Longrightarrow IsSorted[\langle Merge[L, R], n, S\rangle]\Big)$

Property 55 is proved by **Properties 16**, **41**, **37**, and **36**.

Similarly, one can obtain numerous properties which correspond to the sorted permutations of the trees $\langle L, n, Concat[R, S]\rangle$, $\langle Merge[L, R], n, S\rangle$, and $\langle Concat[L, R], n, S\rangle$.

The generation of such properties can also be done automatically. The implementation of the inference rule **IR-4** (see Subsection 3.5.1) uses a mechanism which generates the conjunction of microatoms corresponding to the fact that a certain ground expression is sorted. Moreover, the conjunction is simplified by taking into account currently assumed microatoms, as well as symmetry, and transitivity of ordering relations.

This can be used to partially automate the theory exploratin process. However, we did not pursue this direction of research, because, as explained in the previous subsection, it does not appear to be useful for increasing the proving efficiency.

Further statements which are generated are the definitions of the sorting algorithms and they are presented in Section 5.

5. Algorithm Synthesis Experiments

We present here some practical synthesis experiments realised in the *Theorema* system.

The results presented in this section have been partially published in (Dramnesc et al., 2016b, 2015a, 2016a), see also (Dramnesc et al., 2015b).

5.1. Synthesis of Sort-1

In this subsection, we present the automatically generated proof of **Conjecture 1** in the *Theorema* system. Note that the statement, which has to be proven by induction, is:

$$P[X] : \exists_{T}(X \approx T \wedge IsSorted[T]).$$

Proof: Start to prove **Conjecture 1** using the current knowledge base and by applying **Induction-3**, then **S-1** to eliminate the existential quantifier.

Base case 1: Prove: $\varepsilon \approx T^* \wedge IsSorted[T^*]$.
One obtains the substitution $\{T^* \to \varepsilon\}$ and the new goal is $IsSorted[\varepsilon]$, which is true by **Definition 9**.

Base case 2: Prove: $\langle \varepsilon, n, \varepsilon \rangle \approx T^* \wedge IsSorted[T^*]$.
One obtains the substitution $\{T^* \to \langle \varepsilon, n, \varepsilon \rangle\}$. The new goal is $IsSorted[\langle \varepsilon, n, \varepsilon \rangle]$ which is true by **Definition 9**.

Induction case 1: Assume:

$$\exists_{T}(L_0 \approx T \wedge IsSorted[T]) \tag{1}$$

and prove:

$$\exists_{T}(\langle L_0, n, \varepsilon \rangle \approx T \wedge IsSorted[T]) \tag{2}$$

Apply **S-1** on (1) and (2) to eliminate the existential quantifier. The induction hypothesis becomes:

$$L_0 \approx T_1, \quad IsSorted[T_1] \tag{3}$$

and the goal is:

$$\langle L_0, n, \varepsilon \rangle \approx T^* \wedge IsSorted[T^*] \tag{4}$$

Apply **IR-3** and rewrite our goal (4) by using the first conjunct of the assumption (3). The goal becomes:

$$\langle T_1, n, \varepsilon \rangle \approx T^* \wedge IsSorted[T^*] \tag{5}$$

Apply **IR-4** (to generate permutations of $\langle T_1, n, \varepsilon \rangle$) and **S-3**, then prove the following alternatives:
Alternative-1: One obtains the substitution $\{T^* \to \langle T_1, n, \varepsilon \rangle\}$ to get:

$$IsSorted[\langle T_1, n, \varepsilon \rangle] \tag{6}$$

Apply **IR-1** on (6) and prove:

$$IsSorted[T_1] \wedge RgM[T_1] \leq n \tag{7}$$

Apply **IR-2** using (3) and the new goal is:

$$RgM[T_1] \leq n \tag{8}$$

Apply **IR-5** and the goal (8) becomes the conditional assumption on this branch.
Alternative-2: One obtains the substitution $\{T^* \rightarrow \langle \varepsilon, n, T_1 \rangle\}$. The proof is similar and one has to prove:

$$n \leq LfM[T_1] \tag{9}$$

which becomes the conditional assumption on this branch.
Alternative-3: Since the disjunction of the conditions (8) and (9) is not provable, the prover generates a further alternative. This depends on the synthesis scenario (see the end of Section 2.3). If the properties of the function *Insert* are present in the knowledge base, then the prover generates the substitution $\{T^* \rightarrow Insert[n, T_1]\}$ based on these properties.

If the properties of *Insert* are not present, then the prover generates a failing branch. From the failure, **Conjecture 14** - displayed in Subsection 5.2 - is generated, which is used for the synthesis of *Insert*, as shown in Subsection 5.2. Then, we replay the current proof with knowledge about this auxiliary function and the proof will proceed further.

Induction case 2: Similar to *Induction case 1*, one obtains:
Alternative-1: $\{T^* \rightarrow \langle \varepsilon, n, T_2 \rangle\}$ and the conditional assumption is: $n \leq LfM[T_2]$.
Alternative-2: $\{T^* \rightarrow \langle T_2, n, \varepsilon \rangle\}$ and the conditional assumption is: $RgM[T_2] \leq n$.
Alternative-3: Since the auxiliary function *Insert* is already known, the proof will succeed with the substitution: $\{T^* \rightarrow Insert[n, T_2]\}$.

Induction case 3: Assume:

$$L_1 \approx T_3, \;\; IsSorted[T_3], \;\; R_1 \approx T_4, \;\; IsSorted[T_4] \tag{10}$$

and prove:

$$\langle L_1, n, R_1 \rangle \approx T^* \wedge IsSorted[T^*] \tag{11}$$

Apply **IR-3** and rewrite our goal (11) by using the first and the third conjunct of the assumption (10) and the new goal is:

$$\langle T_3, n, T_4 \rangle \approx T^* \wedge IsSorted[T^*] \tag{12}$$

Apply **IR-4** and **S-3** and obtain the permutations of the list $\langle T_3, n, T_4 \rangle$, for each permutation a number of possible tree expressions as witness for T^*, and for each witness an alternative possibly generating a condition as goal. Note that we can use *Insert* because it was already generated. For instance, the permutation $\langle T_3, n, T_4 \rangle$ generates the tree expression $\langle T_3, n, T_4 \rangle$ with conditions $RgM[T_3] \leq n \leq LfM[T_4]$, as well as the expression $Concat[T_3, Insert[n, T_4]]$ with similar conditions. If the function *Merge* is present in the knowledge base, then also the expression $Merge[T_3, Insert[n, T_4]]$ is generated. The latter does not need conditions, thus the proof succeeds. Note that the first two branches are computationally cheaper, but they can be applied only when the input satisfies certain conditions.

If the function *Merge* is not present, then the branch corresponding to *Concat* will be followed by a failing branch which has the same witness. From this failing branch, we generate **Conjecture 15** which can be used for the synthesis of the function *Merge*.

For the purpose of this presentation, we use only the alternative branch generated by the list $\langle n, T_3, T_4 \rangle$ with expression $Insert[n, Concat[T_3, T_4]]$. This generates the same conjecture for the synthesis of *Merge* and also the last branch in the following sorting algorithm (which was also certified in *Coq*):

$$\underset{n,L,R}{\forall}\left(\begin{array}{c} F_1[\varepsilon] = \varepsilon \\ F_1[\langle \varepsilon, n, \varepsilon\rangle] = \langle \varepsilon, n, \varepsilon\rangle \\ F_1[\langle L, n, \varepsilon\rangle] = \left\{\begin{array}{l} \langle F_1[L], n, \varepsilon\rangle, \text{ if } RgM[F_1[L]] \le n \\ \langle \varepsilon, n, F_1[L]\rangle, \text{ if } n \le LfM[F_1[L]] \\ Insert[n, F_1[L]], \text{ otherwise} \end{array}\right. \\ F_1[\langle \varepsilon, n, R\rangle] = \left\{\begin{array}{l} \langle \varepsilon, n, F_1[R]\rangle, \text{ if } n \le LfM[F_1[R]] \\ \langle F_1[R], n, \varepsilon\rangle, \text{ if } RgM[F_1[R]] \le n \\ Insert[n, F_1[R]], \text{ otherwise} \end{array}\right. \\ F_1[\langle L, n, R\rangle] = Insert[n, Merge[F_1[L], F_1[R]]] \end{array}\right)$$

5.2. Synthesis of Insert *and* Merge

If the necessary properties required for the proof to succeed are missing from the knowledge base (e.g., some auxiliary sub-algorithms are missing), then the proof fails and the new conjecture is generated. Formally, the new conjecture is a universally quantified implication, by transforming back the Skolem constants into universal variables. The left–hand side of the implication consists of the current assumptions and the right–hand side is the failed goal, where the meta–variable becomes an existentially quantified variable. This corresponds to the synthesis problem of a sub-algorithm needed in the main algorithm, following the "**cascading**" principle, described for lists in (Dramnesc and Jebelean, 2015b) and is an extension of the method presented in (Buchberger, 2003). According to this principle, this new synthesis problem is simpler than the original problem because the input has more properties (in our cases, the input trees are sorted). This process of reducing problems into simpler ones can be repeated and is finished when the functions to synthesize are present in the knowledge base. In this case, the whole synthesis process succeeds.

During the synthesis of the algorithm *Sort-1* presented in Subsection 5.1, if the functions *Insert* and *Merge* are not present in the knowledge base, then we construct using (10) and (12) the following conjecture:

Conjecture 13. $\underset{\substack{n,L,R \\ IsSorted[L],IsSorted[R]}}{\forall} \underset{T}{\exists}\Big(\langle L, n, R\rangle \approx T \wedge IsSorted[T]\Big)$

The conjecture is constructed using the following heuristics inspired by (Buchberger and Craciun, 2004): the failing goal is transformed by transforming Skolem constants into universally quantified variables and meta-variables into existentially quantified variables, while the order of the quantifiers is the order in which the corresponding elements have been created during the proof. Moreover, the properties of the Skolem constants are added as conditions for the corresponding quantified variables.

We manually decompose this conjecture into two conjectures which imply it, by using the heuristics:

- separate unknown auxiliary functions in different conjectures;
- decompose a term with many arguments into a composition of functions with fewer arguments;

Formally, the decomposition uses the special property $\langle L, n, R\rangle \approx Concat[L, \langle \varepsilon, n, R\rangle]$.

Conjecture 14. $\underset{\substack{n,R \\ IsSorted[R]}}{\forall} \underset{T}{\exists}\Big(\langle \varepsilon, n, R\rangle \approx T \wedge IsSorted[T]\Big)$

Conjecture 15. $\underset{\substack{L,R\\ IsSorted[L],IsSorted[R]}}{\forall} \underset{T}{\exists}\Big(Concat[L,R] \approx T \wedge IsSorted[T]\Big)$

The following proofs of these conjectures constitute the synthesis of the two auxiliary functions *Insert* and *Merge*.

5.2.1. Synthesis of Insert

The prover automatically generates the proof of **Conjecture 14** by applying **Induction-1** (on the second argument) and the specific inference rules and strategies from Section 3. We describe below the most important steps of the proof. Note that the statement which has to be proven by induction is:

$$P[X] : IsSorted[X] \Longrightarrow (\underset{T}{\exists}(\langle \varepsilon, n, X\rangle \approx T \wedge IsSorted[T])).$$

Proof. After applying **Induction-1** and **S-1** to eliminate the existential quantifier, we get:
Base case: The found witness is $\{T^* \rightarrow \langle \varepsilon, n, \varepsilon\rangle\}$.
Induction step: We assume:

$$IsSorted[L] \Longrightarrow (\langle \varepsilon, n, L\rangle \approx T_1 \wedge IsSorted[T_1]) \tag{13}$$

$$IsSorted[R] \Longrightarrow (\langle \varepsilon, n, R\rangle \approx T_2 \wedge IsSorted[T_2]) \tag{14}$$

and we prove:

$$IsSorted[\langle L, m, R\rangle] \Longrightarrow (\langle \varepsilon, n, \langle L, m, R\rangle\rangle \approx T^* \wedge IsSorted[T^*]) \tag{15}$$

We prove the right–hand side of the above implication, by assuming the left–hand side, which using **IR-1** is decomposed into:

$$IsSorted[L],\ \ IsSorted[R],\ \ RgM[L] \leq m,\ \ m \leq LfM[R],\ \ L \leq m,\ \ m \leq R \tag{16}$$

Using *modus ponens* from (16) by (13) and (14), we obtain:

$$\langle \varepsilon, n, L\rangle \approx T_1,\ IsSorted[T_1],\ \langle \varepsilon, n, R\rangle \approx T_2,\ IsSorted[T_2] \tag{17}$$

The goal is:

$$\langle \varepsilon, n, \langle L, m, R\rangle\rangle \approx T^* \wedge IsSorted[T^*] \tag{18}$$

Since **IR-3a** can be applied on (18) in two different ways, we generate two alternatives:
Alternative-1: By applying **IR-3a** using the first two assumptions from (17), the goal is transformed into:

$$\langle T_1, m, R\rangle \approx T^* \wedge IsSorted[T^*] \tag{19}$$

Obtain substitution $\{T^* \longrightarrow \langle T_1, m, R\rangle\}$ and prove: $IsSorted[\langle T_1, m, R\rangle]$. Apply **IR-1** and the goal becomes:

$$IsSorted[T_1] \wedge IsSorted[R] \wedge RgM[T_1] \leq m \wedge m \leq LfM[R] \tag{20}$$

Eliminate the first two conjuncts of the goal (apply **IR-2** using (17), (16)) and the new goal is: $RgM[T_1] \leq m \wedge m \leq LfM[R]$. Apply **IR-3c** using (17) and the goal becomes: $n \leq m \wedge L \leq m \wedge m \leq LfM[R]$. Apply **IR-2** using (16) and the new goal is: $n \leq m$. This goal fulfils the rule **IR-5** and thus it becomes the conditional assumption on this branch.

Alternative-2: By applying **IR-3a** using the last two assumptions from (17) the new goal is:

$$\langle L, m, T_2\rangle \approx T^* \wedge IsSorted[T^*] \tag{21}$$

Similar to the previous case and by using **Property 14** we obtain the substitution $\{T^* \longrightarrow \langle L, m, T_2\rangle\}$ and the last goal is: $m \leq n$, which becomes the conditional assumption on this branch. □

The synthesized algorithm is:

$$\underset{n,m,L,R}{\forall}\left(\begin{array}{c} Insert[n,\varepsilon] = \langle \varepsilon, n, \varepsilon\rangle \\ Insert[n,\langle L,m,R\rangle] = \left\{\begin{array}{ll} \langle Insert[n,L],m,R\rangle, & \text{if } n \leq m \\ \langle L,m,Insert[n,R]\rangle, & \text{otherwise}\end{array}\right. \end{array}\right) \tag{22}$$

5.2.2. Synthesis of Merge

In a similar manner the synthesis of *Merge* is also performed, by proving **Conjecture 15** using **Induction-1** on the first argument. A significant difference w.r.t. the previous proofs is that this function needs a nested recursion, which cannot be generated by applying only the induction principles presented here. In order to synthesize the algorithm, we use the techniques presented in Subsections 3.3 and 3.4, which are implemented by the inferece rule **IR-4**. Similarly to the situation in the synthesis of *Sort-1* (see proof step after formula (12)), numerous algorithms can be synthesized. We present here one of the algorithms, where 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:

$$\underset{n,L,R,S}{\forall}\left(\begin{array}{r} Merge[\varepsilon,R] = R \\ Merge[\langle L,n,R\rangle,S] = Insert[n,Merge[L,Merge[R,S]]]\end{array}\right) \tag{23}$$

Other different versions of the *Merge* algorithm have been synthesized and detailed in Subsection 5.4.

5.3. Synthesis of Other Sorting Algorithms

5.3.1. Sort-2.

The prover generated automatically the proof of **Conjecture 1** by applying **Induction-2** and by using the current knowledge base (**Definition 9**, **Properties 32**, **41**, and **43** – see Section 4).

The proof is similar with the ones presented above and from this proof the following algorithm is extracted automatically:

$$\underset{n,L,R}{\forall}\left(\begin{array}{c} F_2[\varepsilon] = \varepsilon \\ F_2[\langle L,n,\varepsilon\rangle] = \left\{\begin{array}{l} \langle F_2[L],n,\varepsilon\rangle, \text{ if } RgM[F_2[L]] \leq n \\ \langle \varepsilon,n,F_2[L]\rangle, \text{ if } n \leq LfM[F_2[L]] \\ Insert[n,F_2[L]], \text{ otherwise}\end{array}\right. \\ F_2[\langle L,n,R\rangle] = Merge[F_2[\langle L,n,\varepsilon\rangle],F_2[R]] \end{array}\right)$$

5.3.2. Sort-3.

The proof of **Conjecture 1** is generated automatically by applying **Induction-3** and by using properties from the knowledge base (including properties of *Concat*).

The corresponding algorithm which is extracted automatically from the proof is similar to F_1, excepting the last branch which is:

$$F_3[\langle L,n,R\rangle] = Insert[n,F_3[Concat[L,R]]]$$

5.3.3. Sort-4.

The prover i) automatically generates the proof of **Conjecture 1** by applying **Induction-3** and by using properties from the knowledge base (including properties of *Insert*, *Merge*), and ii) applies the inference rule **IR-4** which generates permutations.

The automatically extracted algorithm is similar to F_1, excepting the last branch for which F_4 has three branches:

$$F_4[\langle L, n, R\rangle] = \begin{cases} \langle F_4[L], n, F_4[R]\rangle, \text{ if } (RgM[F_4[L]] \leq n \ \wedge n \leq LfM[F_4[R]]) \\ \langle F_4[R], n, F_4[L]\rangle, \text{ if } (RgM[F_4[R]] \leq n \ \wedge n \leq LfM[F_4[L]]) \\ Insert[n, Merge[F_4[L], F_4[R]]], \text{ otherwise} \end{cases}$$

5.3.4. Sort-5.

The prover i) generates automatically the proof of **Conjecture 1** by applying **Induction-3** and by using properties from the knowledge base (including properties of *Insert*, *Concat*), and ii) applies the inference rule **IR-4** which generates permutations.

The algorithm which is extracted automatically from the proof is similar to F_3 excepting the last branch, where F_5 has three branches:

$$F_5[\langle L, n, R\rangle] = \begin{cases} \langle F_5[L], n, F_5[R]\rangle, \text{ if } RgM[F_5[L]] \leq n \ \wedge n \leq LfM[F_5[R]] \\ \langle F_5[R], n, F_5[L]\rangle, \text{ if } RgM[F_5[R]] \leq n \ \wedge n \leq LfM[F_5[L]] \\ Insert[n, F_5[Concat[L, R]]], \text{ otherwise} \end{cases}$$

The automatically generated proofs corresponding to these algorithms, their extraction process and the computations with the extracted algorithms in the *Theorema* system are fully presented in (Dramnesc et al., 2015b).

The following table presents the synthesized sorting algorithms. For each of them, **Conjecture 1** has been proved using the induction principles from the second column. The third column shows whether the rule **IR-4** (which generates the permutations and witnesses) is used or not. All the synthesized algorithms use the three auxiliary functions: *LfM*, *RgM*, *Insert* and one of the two functions *Merge* or *Concat* specified in the fourth column.

Extracted algorithm	Induction principle	Uses **IR-4**	Use of *Merge* or *Concat*
F_1	**Induction-3**	No	*Merge*
F_2	**Induction-2**	No	*Merge*
F_3	**Induction-3**	No	*Concat*
F_4	**Induction-3**	Yes	*Merge*
F_5	**Induction-3**	Yes	*Concat*

5.4. Synthesis of Merge *Using Combinatorial Techniques*

The prover automatically generates the proof of **Conjecture 15**, which we present below and which illustrates the combinatorial technique and the lazy induction.

The proof applies **Induction-1** on the first argument of the function *Merge* to be synthesized.

Proof: After applying **Induction-1** and **S-1** to eliminate the existential quantifier, we get:
Base case: The witness found is $\{T^* \rightarrow \langle Concat[\varepsilon, R_0]\}$, which is $\{T^* \rightarrow R_0\}$.
Induction step:

Using strategy **S-1**, after Skolemization of the existential variables into T_1, T_2, the induction hypotheses become:

$$P[L] : \Big((IsSorted[L] \wedge IsSorted[S\,]) \Longrightarrow (Concat[L, S\,] \approx T_1 \wedge IsSorted[T_1])\Big) \tag{24}$$

$$P[R] : \Big((IsSorted[R] \wedge IsSorted[S\,]) \Longrightarrow (Concat[R, S\,] \approx T_2 \wedge IsSorted[T_2])\Big) \tag{25}$$

and the induction goal (to prove) is:

$$P[\langle L, n, R\rangle] : \Big((IsSorted[\langle L, n, R\rangle] \wedge IsSorted[S\,]) \Longrightarrow (Concat[\langle L, n, R\rangle, S\,] \approx T^* \wedge IsSorted[T^*])\Big) \tag{26}$$

where T^* is the meta-variable obtained from the existential variable, for which the prover needs to find a witness term. The right-hand side of the target implication is proven by assuming the left-hand side, which by **IR-1** is decomposed into the following microatoms:

$$IsSorted[L] \tag{27}$$

$$IsSorted[R] \tag{28}$$

$$L \preceq n \tag{29}$$

$$n \preceq R \tag{30}$$

$$L \ll R \tag{31}$$

$$IsSorted[S\,] \tag{32}$$

Using *modus ponens* from (24) and (25) by (27) and (28), further assumptions are obtained:

$$Concat[L, S\,] \approx T_1 \tag{33}$$

$$IsSorted[T_1] \tag{34}$$

$$Concat[R, S\,] \approx T_2 \tag{35}$$

$$IsSorted[T_2] \tag{36}$$

The goal is:

$$Concat[\langle L, n, R\rangle, S\,] \approx T^* \wedge IsSorted[T^*] \tag{37}$$

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 $Merge[\langle L, n, R\rangle, S\,]$.)

Since **IR-3b** can be applied on (37) in two different ways, we generate two alternatives:
Alternative-1: By applying **IR-3b** using (33), the goal is transformed into:

$$\langle T_1, n, R\rangle \approx T^* \wedge IsSorted[T^*] \tag{38}$$

At 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, generates all possible expressions containing all the symbols in the list, composed by using the tree constructors and

the functions *Concat*, *Insert*, 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:

$$\{RgM[T_1] \leq n\} \Longrightarrow \langle T_1, n, R\rangle$$

$$\{RgM[T_1] \leq LfM[R], RgM[T_1] \leq n\} \Longrightarrow Concat[T_1, Insert[n, R]]$$

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

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

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

$$\{\} \Longrightarrow Merge[T_1, Insert[n, R]]$$

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

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

$$\{RgM[R] \leq LfM[T_1]\} \Longrightarrow Insert[n, Concat[R, T_1]]$$

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

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 $Merge[t_2, t_1]$ if the expression of another clause contains $Merge[t_1, t_2]$ in a similar expression at the same level. This simplification is performed because the function *Merge* is symmetric modulo the equivalence relation $\approx$.

The following simplifications are also applied because they improve the respective expres-

sions from the computational point of view:

$$\begin{aligned} Merge[\langle \varepsilon, n, \varepsilon \rangle, X] &\longrightarrow Insert[n, X], \\ Merge[X, \langle \varepsilon, n, \varepsilon \rangle] &\longrightarrow Insert[n, X], \\ Concat[\langle L, n, \varepsilon \rangle, R] &\longrightarrow \langle L, n, R \rangle. \end{aligned} \tag{39}$$

Each expression is further processed by replacing each occurrence of T_1 by $Merge[L, S\,]$ — according to (24), and also by replacing the conditions involving T_1 with the appropriate conditions involving L, 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).

$$\{\} \Longrightarrow Merge[\langle \varepsilon, n, R \rangle, Merge[L, S\,]] \tag{40}$$
$$\{\} \Longrightarrow Merge[Insert[n, R], Merge[L, S\,]] \tag{41}$$
$$\{\} \Longrightarrow Insert[n, Merge[Merge[L, S\,], R]] \tag{42}$$
$$\{\} \Longrightarrow Merge[Insert[n, Merge[L, S\,]], R] \tag{43}$$
$$\{S \preceq n\} \Longrightarrow \langle Merge[L, S\,], n, R \rangle \tag{44}$$
$$\{S \ll R\} \Longrightarrow Insert[n, Concat[Merge[L, S\,], R]] \tag{45}$$
$$\{S \preceq n\} \Longrightarrow Merge[\langle Merge[L, S\,], n, \varepsilon \rangle, R] \tag{46}$$
$$\{S \preceq n\} \Longrightarrow Concat[Merge[L, S\,], Insert[n, R]] \tag{47}$$

Note that the clauses (40), (41), (43) do not fulfil the termination criterion: the first recursive call to $Merge$ has the same multiset of symbols as the main call $Merge[\langle L, n, R \rangle], S\,]$.

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 ensured only for (42). Additionally, the algorithm may contain one of clauses (44), (46), (47), conditioned by $S \preceq n$ and may contain the clause (45) 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 *Insert, 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 (44), (45), (42) (in this order):

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

Alternative-2: By applying **IR-3b** using (35), the goal is transformed into:

$$\langle L, n, T_2 \rangle \approx T^* \wedge IsSorted[T^*] \tag{48}$$

The proof proceeds similarly as in ***Alternative-1*** and similar cases are generated, the only difference consists in using the recursive call $Merge[R, S\,]$ instead of $Merge[L, S\,]$ as in ***Alternative-1***.

The list of the clauses obtained after all the simplification steps are:

$$\{\} \Longrightarrow Merge[Insert[n, L], Merge[R, S]] \tag{49}$$
$$\{\} \Longrightarrow Merge[\langle L, n, \varepsilon\rangle, Merge[R, S]] \tag{50}$$
$$\{\} \Longrightarrow Insert[n, Merge[L, Merge[R, S]]] \tag{51}$$
$$\{\} \Longrightarrow Merge[Insert[n, Merge[R, S]], L] \tag{52}$$
$$\{L \ll S\} \Longrightarrow \langle Insert[n, Concat[L, Merge[R, S]]] \tag{53}$$
$$\{L \ll S\} \Longrightarrow Concat[L, Insert[n, Merge[R, S]]] \tag{54}$$
$$\{n \preceq S\} \Longrightarrow Merge[\langle \varepsilon, n, Merge[R, S]\rangle, L] \tag{55}$$
$$\{n \preceq S\} \Longrightarrow \langle L, n, Merge[R, S]\rangle \tag{56}$$

The most important clause generated here and which is the analogous of (42), is (51).

Any algorithm composed from such conditional clauses must fulfil two important conditions: *proper ordering of clauses* and *coverage of all cases*. One possible algorithm is based on clauses (56),(54),(51) (in this order):

$$\underset{n,L,R,S}{\forall} \left(Merge[\langle L, n, R\rangle, S] = \begin{cases} Merge[\varepsilon, S] = S \\ \langle L, n, Merge[R, S]\rangle, \text{ if } n \preceq LfM[S] \\ Concat[L, Insert[n, Merge[R, S]]], \\ \qquad \text{if } RgM[L] \preceq LfM[S] \\ Insert[n, Merge[L, Merge[R, S]]], \text{ otherwise} \end{cases} \right)$$

'Proper ordering of clauses' means that a clause which is more general – like e.g. (42) must be placed after the ones which are less general – like e.g. after (45), 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 the 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 list, 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: i) each set of conditions is completed with the conditions from the current proof assumptions and with all the transitive consequences; ii) all sets of conditions (as conjunctions) are composed into a disjunction, and its CNF is computed; iii) for validity, each disjunctive clause must be valid, therefore it must contain both $a \preceq b$ and $b \preceq a$, 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.

6. Additional Certification of the Synthesized Algorithms

The theoretical basis and the correctness of the presented proof-based synthesis scheme is well known – see for instance (Bundy et al., 2006). However, the implementation of the rules in *Theorema* is error-prone. To check the soundness of the implementation, we have mechanically verified that the synthesized algorithms satisfy the correctness condition, by using the *Coq* proof assistant (`https://coq.inria.fr`). In order to do this, we have changed the specification of several functions and add new definitions. The proofs have been conducted mostly interactively, by using explicit induction principles whose induction schemas derive from the analysis of recursive data structures and function definitions, as well as new lemmas and proof techniques.

The *Coq* specification. The binary tree data structure can be represented in *Coq* as an inductive set.

```
Inductive Btree : Set :=
  | Epsilon : Btree
  | Node : Btree → nat → Btree → Btree.
```

Here, **nat** is the *Coq* data structure for naturals.

Since *Coq* requires that any function be total, the definition of *LfM* and *RfM* functions has slightly changed from the partial definitions given here. The conversion into total functions is possible if the components of the triplet given as argument are represented as the new arguments.

```
Fixpoint rgm (l:Btree) (n:nat) (r:Btree): nat :=
match r with
  | Epsilon ⇒ n
  | Node t1 m t2 ⇒ rgm t1 m t2
end.

Fixpoint lfm (l:Btree) (n:nat) (r:Btree): nat :=
  match l with
    | Epsilon ⇒ n
    | Node t1 m t2 ⇒ lfm t1 m t2
  end.
```

The case analysis on the values of some of their arguments is performed by the means of the `match ... with` construction. In order to simplify the definition of certain lemmas, the equivalent representations of *LfM* and *RfM* from the paper, given below by the definitions of the minimal and maximal functions, respectively (intended to return the minimal and maximal elements of sorted trees), are used, under the condition that their argument is never Epsilon. Hence, in these definitions, the value returned when the argument is Epsilon can be any natural number (in our case, it is 0).

```
Definition maximal (t:Btree):nat:=
  match t with
  | Epsilon ⇒ 0
  | Node l n r ⇒ rgm l n r
  end.
```

```
Definition minimal (t:Btree):nat:=
  match t with
  | Epsilon ⇒ 0
  | Node l n r ⇒ lfm l n r
  end.
```

The number of elements (or *nodes*) from a binary tree, denoted by nr_nodes, serves also to prove the termination of some recursive definitions, for example F_3.

```
Fixpoint nr_nodes (t: Btree) : nat :=
  match t with
    | Epsilon ⇒ 0
    | Node tl n tr ⇒ S ((nr_nodes tl) + (nr_nodes tr))
  end.
```

The recursive definition of F3 requires the use of the `Function` tool (Balaa and Bertot, 2000; Barthe and Courtieu, 2002; Barthe et al., 2006). Its termination can be proved using a well-founded ordering in terms of the anonymous function that follows the `wf` keyword, which compares the number of nodes of the two trees given as argument.

```
Function F3 (t: Btree) {wf (fun t1 t2: Btree ⇒ (nr_nodes t1) < ( nr_nodes t2)) t}: Btree :=
  match t with
    | Epsilon ⇒ Epsilon
    | Node Epsilon n Epsilon ⇒ Node Epsilon n Epsilon
    | Node ((Node a n1 b) as tl) n Epsilon ⇒
      match F3 tl with
        | Epsilon ⇒ Epsilon
        | (Node a1 z1 b1) as tr ⇒ if leq (rgm a1 z1 b1) n then Node tr n Epsilon else if
leq n (lfm a1 z1 b1) then Node Epsilon n tr else insert n (merge tr Epsilon)
      end
    | Node Epsilon n ((Node c b d) as tr) ⇒
      match F3 tr with
        | Epsilon ⇒ Epsilon
        | (Node a2 z2 b2) as tl' ⇒if leq n (lfm a2 z2 b2) then Node Epsilon n tl' else if
leq (rgm a2 z2 b2) n then Node tl' n Epsilon else insert n (merge tl' Epsilon)
      end
    | Node ((Node a n1 a2) as tl) n ((Node c b d) as tr) ⇒ insert n (F3 (concat tl tr))
  end.
```

Here, leq is the 'less or equal' predicate over naturals.

The definition of F3 is accepted by *Coq* if the termination conditions, stating that the arguments of F3 in the body of the definition are smaller (w.r.t. the well-founded ordering) than its argument, are proved.

We have encountered some difficulties with the acceptation of the F5 definition by *Coq*, since the `Function` tool does not support nested recursion or constructions that may allow nested recursion. Or, the last branch of F5 algorithm includes a nested `match` F5 ... `with` construction that may allow nested recursion, but in our case it is not nested recursion. However, the definition of F5 can be accepted by *Coq* if all the recursive calls from this branch are arguments of some 'inline' function, here denoted by combine_results. By unfolding the inline function, one can get the original definition of F5.

```
Definition combine_results (n:nat) (t1 t2 t3: Btree) :=
  match t1 with
  | Epsilon ⇒
    match t2 with
    | Epsilon ⇒ Node Epsilon n Epsilon
    | Node l3 n3 r3 as tr' ⇒
      if leq n (lfm l3 n3 r3) then Node Epsilon n tr'
      else if leq (rgm l3 n3 r3) n then Node tr' n Epsilon else insert n t2
    end
  | Node l4 n4 r4 as tl'⇒
    match t2 with
    | Epsilon ⇒
      if leq (rgm l4 n4 r4) n then Node tl' n Epsilon
      else if leq n (lfm l4 n4 r4) then Node Epsilon n tl' else insert n t1
    | Node l3 n3 r3 as tr' ⇒
      if andb (leq n (lfm l3 n3 r3)) (leq (rgm l4 n4 r4) n) then Node tl' n tr' else
        if andb (leq n (lfm l4 n4 r4)) (leq (rgm l3 n3 r3) n) then Node tr' n tl' else
          insert n t3
    end
  end.
```

The definition of F5 can be obtained from that of F3 by replacing F3 with F5 and the right-hand side expression of the last ⇒ symbol, i.e., insert *n* (F3 (concat *tl tr*)), with combine_results *n* (F5 *tl*) (F5 *tr*) (F5 (concat *tl tr*)).

The specification also includes four new functions representing basic operations on elements, lists of elements and trees, presented in the next subsection.

The *Coq* proofs. The proof effort was non-trivial, involving significant user interaction. Most of the induction steps are based on structural induction schemas issued from the definition of inductive sets for naturals, lists and binary trees, that use an induction ordering different from that used in the paper. On the other hand, the proofs of the **Conjecture 1** for F3 and F5 require induction schemas issued from the recursive definition of F3 and F5, respectively. They use a similar induction ordering as that defined in the paper, in terms of the anonymous function that follows the `wf` keyword. However, the number of induction cases equals the number of leaves in the tree structure of the algorithm definition. In our case, there are 11 induction cases issued from both algorithms, hence these induction schemas are different from the induction schemas presented in the paper.

Also, 35 out of the 50 used lemmas are new. It is worth to explain the difficulties we have encountered to prove one of the crucial lemmas, denoted by lfmrgm and stating that the leftmost element in a non-empty sorted tree is smaller or equal than the root element which, in its turn, is smaller or equal than the rightmost element.

```
Lemma lfmrgm: ∀ l r n, isSorted (Node l n r) = true → leq (lfm l n r) n = true ∧ leq n (rgm l n r)
= true.
```

It can be noticed that the proof does not succeed if the previous induction schemas are applied either on *l* or *r*, mainly because, for some induction cases, the induction hypotheses are not strong

enough to prove the corresponding induction conclusions. Our solution was to generalize this lemma, by a new lemma denoted by lfmrgm_generalized and stating that the leftmost element in a non-empty sorted tree is smaller or equal than any element in the tree (among which the root element) which, in its turn, is smaller or equal than the rightmost element.

```
Lemma lfmrgm_generalized: ∀ t, ∀ m, t ≠ Epsilon → isSorted (t) = true → lmember m
(lnodes t) = true → leq (minimal t) m = true ∧ leq m (maximal t) = true.
```

Here, the lnodes function takes as argument a tree and returns the list of nodes in the tree.

```
Fixpoint lnodes (t:Btree) : list nat :=
  match t with
    | Epsilon ⇒ nil
    | Node tl n tr ⇒ lconc (lnodes tl) (lconc [n] (lnodes tr))
  end.
```

The lconc function concatenates two lists of elements.

```
Fixpoint lconc (l: list nat) (l': list nat) : list nat :=
  match l with
      | nil ⇒ l'
      | a :: tl ⇒ a :: (lconc tl l')
  end.
```

The lmember function takes an element and a list of elements and returns true iff the element is in the list.

```
Fixpoint lmember (n:nat) (l: list nat) : bool :=
  match l with
    | nil ⇒ false
    | h :: tl ⇒ if eq_nat1 n h then true else lmember n tl
  end.
```

Here, eq_nat1 takes two naturals and returns true iff they are equal.

```
Fixpoint eq_nat1 (n1:nat) (n2:nat) : bool :=
  match n1,n2 with
    | 0,0 ⇒ true
    | (S n), (S p) ⇒ eq_nat1 n p
    | _,_ ⇒ false
  end.
```

The proof of lfmrgm_generalized succeeds by applying the structural induction on t. It can also be noticed that the induction reasoning, performed on the version using the explicit representation of non-empty trees, fails for similar reasons as for lfmrgm:

```
Lemma lfmrgm_generalized_fail: ∀ l r n, ∀ m, isSorted (Node l n r) = true → lmember m
(lnodes (Node l n r)) = true → leq (lfm l n r) m = true ∧ leq m (rgm l n r) = true.
```

When possible, we have chosen the implicit representation of non-empty trees in the definition of certain lemmas, by adding the condition of the form $t \neq$ Epsilon instead of using (Node l n r). The proofs of the lemmas using $t \neq$ Epsilon are shorter because only one induction step is required (on t), while the proofs of the lemmas using (Node l n r) may require induction steps

for both l and r.

The inference rules and proof strategies are completely different from those generating the synthesized algorithms. On the other hand, several new lemmas are already properties from the theory of binary trees presented in Section 4, for example the transitivity of leq (see **Property** 1). Other lemmas are just the *Coq* equivalent of some of other properties, bridging definitions from *Coq* and the theory of binary trees. For example, **Property** 19 uses the *Member* predicate (which checks if an element is in a binary tree). This property can also be expressed in *Coq* by the means of lnodes and lmember, as below

```
Lemma memberRoot: ∀ l n r, lmember n (lnodes (Node l n r)) = true.
```

The full *Coq* script is available at: http://web.info.uvt.ro/~idramnesc/JSC2016/coq.zip.

7. Related Work

We survey here the main approaches to proof–based algorithm synthesis and compare them to our work.

7.1. Theory Exploration

We extend the work presented in (Buchberger, 2000), where the author explains how to explore a theory in a systematic way. The approach is further refined in (Buchberger, 2004), where a scheme-based model for theory exploration in the *Theorema* system is introduced. This is also applied on natural numbers in (Craciun and Hodorog, 2007). Theory exploration consists of two processes: top–down and bottom–up, which are followed in parallel. The top–down exploration starts from the main concepts which have to be defined, and then introduces hierarchically more auxiliary concepts. The bottom–up exploration starts from the axioms and definitions of the basic concepts and proceeds by exploring the relations between them in a systematic manner. In our experiments we follow the same approach, additionally including new definitions and properties when they appear to be useful in the proofs which are performed for synthesis purposes.

There are numerous research results concerning *automation of theory exploration*. In (Colton, 2012) we find a comprehensive discussion of various theoretical and practical aspects of theory exploration, a survey of various tools for computer–aided theory exploration, as well as the description of the system HR for the automation of discovery of mathematical theories. Higher–level scheme based invention of conjectures and definitions is successfully used in (Montano-Rivas et al., 2012) for creating automatically theories on numbers and on lists. An interesting contribution based on systematic creation of conjectures is presented in (Claessen et al., 2013) (we discuss this in the next subsection).

The theory formation for recursive data structures can also be *generative*, as in the approach implemented in the IsaCoSy tool (Johansson et al., 2011). By following some heuristics, IsaCoSy can generate conjectures built from a signature based on a given set of recursive functions and data structures. Integrated into the Isabelle proof environment, IsaCoSy helped to build theories about naturals, lists and trees by calling the proof-planner IsaPlanner (Dixon and Fleuriot, 2003; Dixon and Johansson, 2007) during the attempts to prove conjectures, or a counterexample checker to disprove conjectures. IsaPlanner uses rippling-based heuristics to guide the generation of induction proofs.

A more related tool, using a deductive theory formation approach, is the MATHsAiD system (McCasland and Bundy, 2006). It is able to perform forward reasoning to *deduce* conjectures, rather than generating them, from a set of *terms of interest* carefully built to limit their size. Once a conjecture is deduced, one of its variables is replaced by a term of size two or three, i.e., it is either $S(S(0))$ or $(S(S(S(0))))$ if it is a natural variable, and it is either of the form $[a,b]$ or $[a,b,c]$ if the variable is a list. If the conjecture holds for these values, there is hope that it holds for any value of the variable.

Most of the work on theory exploration is on developing systematic techniques for the automation of the process. We do not discuss this research in more detail because our research is not focussed on the *automation* of theory exploration, but on proof automation.

In Subsections 4.3.1 and 4.3.2 we discuss a method to generate automatically certain type of properties, based on equivalence and ordering. This method has similarities to the method presented in (Claessen et al., 2013). However, for the purpose of proof automation, it was not efficient to generate a large number of properties, but rather to generate appropriate terms and atoms related to the current proof situation.

7.2. Induction Reasoning

The induction reasoning used by the current reasoning tools, e. g. IsaPlanner and MATHsAiD (McCasland and Bundy, 2006), is *eager* and based on explicit induction schemas, requiring the definition of induction hypotheses before their real use in the proof. Indeed, some induction hypotheses may be defined but not used, or it may happen that crucial induction hypotheses are lacking. In MATHsAiD, the variable instantiated by a term of size two or three is the induction variable used during the explicit induction reasoning performed for proving that the conjecture holds for any value of the variable. This comes into contrast with our approach, where explicit induction schemas are discovered during the proofs, by detecting subgoals which have the same structure as the original conjecture, but contain terms which are smaller with respect to a certain Noetherian ordering (Subsection 3.4).

A powerful technique for discovering new implicit induction schemas is *rippling* described in (Bundy et al., 2006). In the current stage of our experiments such sofistication was not necessary, because the methods we used have been able to prove all conjectures and properties from the theory at hand.

The combinatorial technique used for generating possible witness terms scans all possible combinations of constants and appropriate function symbols, therefore it could benefit from some ideas developed for the automatic generation of theories (Johansson et al., 2011; McCasland and Bundy, 2006). However, since in our experiments this generation takes only few seconds, we did not investigate yet possible improvements of the straightforward approach. Our method is close to the one in (Claessen et al., 2013), where new terms and equational conjectures between them are also generated systematically — sometimes by thousands, and then filtered by testing and proving. In our experiments we only need terms involving certain function symbols and certain constants, as explained in Subsection 3.3, their numbers are in the range of hundreds and by simplification they reduce to dozens. Therefore the space/time consumption is negligeable compared to the rest of the synthesis proofs.

7.3. Algorithm Synthesis

An overview of the most common approaches used to tackle the synthesis problem is presented in (Gulwani, 2010).

The proof environments underlying deductive synthesis frameworks are usually supporting both automated and interactive proof methods. Those based on abstract datatype and computation refinements (Wirth, 1971; Back and von Wright, 1998) integrate techniques that are mainly executed manually and implemented by higher-order proof assistants like Isabelle/HOL (Nipkow et al., 2002) or more synthesis-oriented tools as Specware (Smith, 2005). On the other hand, automated proof steps can be performed with decision procedures, e.g., for linear arithmetics, or SAT and SMT solvers as those integrated in Leon (Kneuss et al., 2013). The generated algorithms can be checked for conformity with the input specification by validating the proof trails for each refinement process, for example using the *Coq* library Fiat (Delaware et al., 2015) to ensure the soundness of the validation step by certification with the *Coq* kernel. (Cohen et al., 2013) presents a different *Coq* library using datatype refinement to verify parameterized algorithms for which the soundness proof of some version can be deduced from that of a previous (less efficiently implemented) version. In (Basin et al., 2004) we find a comparison between three synthesis methods: constructive/deductive synthesis, schema-based synthesis and inductive synthesis. This complements the survey of logic program synthesis which has been done in (Deville and Lau, 1994) and (Flener, 2002).

The focus of the previous work on algorithm synthesis is mainly on the classification of the synthesis methods based on different principles of direct synthesis. The focus of our work is on experimental proof–based synthesis, based on the principles presented e.g. in (Bundy et al., 2006). In these experiments we aim at finding effective and efficient proof techniques.

8. Conclusions and Further Work

Our results are: a new theory of binary trees, an arsenal of special strategies and specific inference rules based on properties of binary trees, a new prover in the *Theorema* system which generates all the presented synthesis and theory exploration proofs, an extractor in the *Theorema* system which is able to extract from a proof the corresponding algorithms (including `if-then-else` structures), the synthesis of five sorting algorithms and several versions of their auxiliary functions. We have also certified by *Coq* the soundness property of all five sorting algorithms with the current implementation of the auxiliary functions. The certification proofs are more complex and their generation less automatic than for the *Theorema* proofs that helped for extracting the sorting algorithms, by using different inference rules and additional properties.

The problem of sorting binary trees does not appear to have an important practical significance. It appears more efficient to construct the corresponding list, then sort the list and then construct the tree. The algorithms which we synthesize are not the most efficient, but the nice part is that we work on the same data structure. This case study on synthesis of algorithms and theory exploration is a very interesting exercise from the point of view of finding efficient proof–based methods and techniques for the domain of binary trees. As further work, we want to use the method presented in this paper on more complex recursive data structures (e.g. red-black trees).

Some of the benefits of using the combinatorial techniques presented in this paper are: the generation of numerous witnesses for an existential goal which lead to discovering new recursion structures that are not given from the beginning by the induction principle, discovering induction schemas, discovering algorithms with nested recursion.

The processes of synthesis and verification of the sorting algorithms need a correct formalization of the domain of binary trees. We show that the construction of such a correct and sufficient formalization is possible and it is not a trivial task. The theory contains a very large number of

notions and properties necessary in the process of proof–based synthesis and in the process of algorithm certification.

Our experiments in the *Theorema* system show how one can discover numerous algorithms for the same functions, differing in efficiency and complexity if one applies different induction principles and chooses different alternatives in the proofs. The certification in the *Coq* system of the synthesized algorithms can be seen as a test for checking the soundness of our approach. An alternative to avoid the certification step is the generation of the proofs and the implementation of the inference rules and strategies directly in *Coq*. This would ensure that every synthesized algorithm by these inference rules and strategies is implicitly sound. Some downsides of this approach are its difficulty to prove the soundness of the inference system and the inadequacy for rapid prototyping and testing new ideas. A reasonable compromise would be to devise procedures for translating the *Theorema* proofs directly into *Coq* scripts, by following similar translation procedures as those used for implicit induction proofs (Stratulat, 2017).

Acknowledgments

We thank Yves Bertot for showing us the trick that fools the `Function` tool by using 'inline' functions. We also thank the annonymous referees for their suggestions to improve the quality of the paper.

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

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press.

Back, R.-J., von Wright, J., 1998. Refinement Calculus. Springer Verlag.

Balaa, A., Bertot, Y., 2000. Fix-point equations for well-founded recursion in type theory. In: Aagaard, M., Harrison, J. (Eds.), Theorem Proving in Higher Order Logics. Vol. 1869 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1–16.

Barthe, G., Courtieu, P., 2002. Efficient reasoning about executable specifications in Coq. In: Theorem Proving in Higher Order Logics. Vol. 2410 of LNCS. Springer Berlin, pp. 31–46.

Barthe, G., Forest, J., Pichardie, D., Rusu, V., 2006. Defining and reasoning about recursive functions: A practical tool for the *Coq* proof assistant. In: Hagiya, M., Wadler, P. (Eds.), Functional and Logic Programming. Vol. 3945 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 114–129.

Basin, D., Deville, Y., Flener, P., Hamfelt, A., Nilsson, J. F., 2004. Synthesis of programs in computational logic. In: Program Development in Computational Logic. Springer, pp. 30–65.

Bertot, Y., Casteran, P., 2004. Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions. Vol. XXV of Texts in Theoretical Computer Science. Springer.

Buchberger, B., 2000. Theory Exploration with Theorema. In: Analele Universitatii Din Timisoara, Ser. Matematica-Informatica. Vol. XXXVIII. pp. 9–32.

Buchberger, B., 2003. Algorithm invention and verification by lazy thinking. In: Analele Universitatii de Vest, Timisoara, Ser. Matematica - Informatica. Vol. XLI. pp. 41–70.

Buchberger, B., 2004. Algorithm supported mathematical theory exploration: A personal view and strategy. In: Proceedings of AISC 2004. Vol. 3249 of Springer LNAI. pp. 236 – 250.

Buchberger, B., Craciun, A., 2004. Algorithm synthesis by lazy thinking: Examples and implementation in Theorema. Electr. Notes Theor. Comput. Sci. 93, 24–59.

Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., Windsteiger, W., 2016. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. JFR 9 (1), 149–185.

Bundy, A., Dixon, L., Gow, J., Fleuriot, J., March 2006. Constructing induction rules for deductive synthesis proofs. Electron. Notes Theor. Comput. Sci. 153, 3–21.

Claessen, K., Johansson, M., Rosén, D., Smallbone, N., 2013. Automating Inductive Proofs Using Theory Exploration. In: CADE-24: Proceedings of the 24th International Conference on Automated Deduction. Springer Berlin Heidelberg, pp. 392–406.

Cohen, C., Dénès, M., Mörtberg, A., 2013. Refinements for free! In: Gonthier, G., Norrish, M. (Eds.), Certified Programs and Proofs. Vol. 8307 of Lecture Notes in Computer Science. Springer International Publishing, pp. 147–162.

Colton, S., 2012. Automated Theory Formation in Pure Mathematics. Springer Science & Business Media.

Craciun, A., Hodorog, M., 2007. Decompositions of Natural Numbers: From A Case Study in Mathematical Theory Exploration. In: SYNASC 2007: Proceedings of the 8th International Symposium on Numeric and Symbolic Algorithms for Scientific Computing. IEEE, pp. 41 – 47.

Delaware, B., Claudel, C. P., Gross, J., Chlipala, A., 2015. Fiat: Deductive synthesis of abstract data types in a proof assistant. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '15. ACM, New York, NY, USA, pp. 689–700.

Deville, Y., Lau, K. K., 1994. Logic program synthesis. J. Log. Program. 19/20, 321–350.

Dixon, L., Fleuriot, J. D., 2003. IsaPlanner: A prototype proof planner in Isabelle. In: Proceedings of CADE'03. Vol. 2741 of LNCS. pp. 279–283.

Dixon, L., Johansson, M., 2007. Isaplanner 2: A proof planner in Isabelle. DReaM Technical Report (System description).

Dramnesc, I., Jebelean, T., 2011. Proof techniques for synthesis of sorting algorithms. In: SYNASC 2011: Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE Computer Society, pp. 101–109.

Dramnesc, I., Jebelean, T., 2012a. Automated synthesis of some algorithms on finite sets. In: SYNASC 2012: Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE Computer Society, pp. 143 – 151.

Dramnesc, I., Jebelean, T., 2012b. Discovery of inductive algorithms through automated reasoning: a case study on sorting. In: SISY 2012: Proceedings of IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics. IEEE Xplore, pp. 293 – 298.

Dramnesc, I., Jebelean, T., 2012c. Theory exploration in Theorema: Case study on lists. In: SACI 2012: Proceedings of IEEE 7th International Symposium on Applied Computational Intelligence and Informatics. IEEE Xplore, pp. 421 – 426.

Dramnesc, I., Jebelean, T., 2015a. A case study in proof based synthesis of algorithms on monotone lists. In: SACI 2015: Proceedings of the 10th IEEE International Symposium on Applied Computational Intelligence and Informatics. IEEE, pp. 483 – 488.

Dramnesc, I., Jebelean, T., 2015b. Synthesis of list algorithms by mechanical proving. Journal of Symbolic Computation 68, 61–92.

Dramnesc, I., Jebelean, T., Stratulat, S., 2015a. Combinatorial techniques for proof-based synthesis of sorting algorithms. In: SYNASC 2015: Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. pp. 137 – 144.

Dramnesc, I., Jebelean, T., Stratulat, S., 2015b. Synthesis of some algorithms for trees: Experiments in *Theorema*. Tech. Rep. 15-04, RISC Report Series, Johannes Kepler University, Linz, Austria.

Dramnesc, I., Jebelean, T., Stratulat, S., 2015c. Theory exploration of binary trees. In: SISY 2015: Proceedings of the 13th IEEE International Symposium on Intelligent Systems and Informatics. IEEE, pp. 139 – 144.

Dramnesc, I., Jebelean, T., Stratulat, S., 2016a. A case study on algorithm discovery from proofs: The insert function on binary trees. In: SACI 2016: Proceedings of the 11th IEEE International Symposium on Applied Computational Intelligence and Informatics. IEEE, pp. 231 – 236.

Dramnesc, I., Jebelean, T., Stratulat, S., 2016b. Proof-based synthesis of sorting algorithms for trees. In: LATA 2016: Proceedings of the 10th International Conference on Language and Automata Theory and Applications. Springer, pp. 562 – 575.

Flener, P., 2002. Achievements and prospects of program synthesis. In: Computational Logic: Logic Programming and Beyond. pp. 310–346.

Gulwani, S., 2010. Dimensions in program synthesis. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. PPDP '10. ACM, New York, NY, USA, pp. 13–24.

Johansson, M., Dixon, L., Bundy, A., 2011. Conjecture synthesis for inductive theories. Journal of Automated Reasoning 47 (3), 251–289.

Kneuss, E., Kuraj, I., Kuncak, V., Suter, P., 2013. Synthesis modulo recursive functions. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '13. ACM, New York, NY, USA, pp. 407–426.

Knuth, D. E., 1998. The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition. Addison Wesley Longman Publishing, Redwood City, CA, USA.

Konev, B., Jebelean, T., October 2001. Solution lifting method for handling meta-variables in Theorema. In: Maruster, S.,

Buchberger, B., Negru, V., Jebelean, T. (Eds.), Proceedings of SYNASC01. Mirton, Timisoara, Romania, pp. 15–23.
Kowalski, R., Kuehner, D., 1971. Linear resolution with selection function. Artificial Intelligence 2.
McCasland, R. L., Bundy, A., 2006. MATHsAiD: A mathematical theorem discovery tool. In: Negru, V., Petcu, D., Zaharie, D., Abraham, A., Buchberger, B., Cicortas, A., Gorgan, D., Quinqueton, J. (Eds.), 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2006), 26-29 September 2006, Timisoara, Romania. IEEE Computer Society, pp. 17–22.
Montano-Rivas, O., McCasland, R., Dixon, L., Bundy, A., 2012. Scheme-based Theorem Discovery and Concept Invention. Expert Systems with Applications 39 (2), 1637–1646.
Mordechai, B., 2004. Mathematical Logic for Computer Science. Springer.
Nipkow, T., Paulson, L. C., Wenzel, M., 2002. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Vol. 2283 of Lecture Notes in Computer Science. Springer.
Pelletier, F. J., 2000. A history of natural deduction and elementary logic textbooks. Logical consequence: Rival approaches 1, 105–138.
Smith, D. R., 2005. Generating programs plus proofs by refinement. In: Meyer, B., Woodcock, J. (Eds.), Verified Software: Theories, Tools, Experiments, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Vol. 4171 of Lecture Notes in Computer Science. Springer, pp. 182–188.
Stratulat, S., 2012. A unified view of induction reasoning for first-order logic. In: Voronkov, A. (Ed.), Turing-100 (The Alan Turing Centenary Conference). Vol. 10 of EPiC Series. EasyChair, pp. 326–352.
Stratulat, S., 2017. Mechanically certifying formula-based Noetherian induction reasoning. Journal of Symbolic Computation 80, Part 1, 209–249.
Wirth, N., Apr. 1971. Program development by stepwise refinement. Commun. ACM 14 (4), 221–227.
Wolfram, S., 2003. The Mathematica Book. Wolfram Media Inc.