AlCons: Deductive Synthesis of Sorting Algorithms in *Theorema*

Isabela Drămnesc[1][0000−0003−4686−2864]
Tudor Jebelean[2][0000−0002−2247−2151]

[1] Department of Computer Science, West University Timişoara, Romania
Isabela.Dramnesc@e-uvt.ro
https://staff.fmi.uvt.ro/~isabela.dramnesc
[2] RISC, Johannes Kepler University, Linz, Austria
Tudor.Jebelean@jku.at
https://risc.jku.at/m/tudor-jebelean

Abstract. We describe the principles and the implementation of ***AlCons*** (*Algorithm Constructor)*, a system for the automatic proof–based synthesis of sorting algorithms on lists and on binary trees, in the frame of the *Theorema* system. The core of the system is a dedicated prover based on specific inference rules and strategies for constructive proofs over the domains of lists and of binary trees, aimed at the automatic synthesis of sorting algorithms and their auxiliary functions from logical specifications. The specific distinctive feature of our approach is the use of multisets for expressing the fact that two lists (trees) have the same elements. This allows a more natural expression of the properties related to sorting, compared to the classical approach using the permutation relation (a list is a permutation of another). Moreover, the use of multisets leads to special inference rules and strategies which make the proofs more efficient, as for instance: expand/compress multiset terms and solve meta-variables using multiset equalities. Additionally we use a Noetherian induction strategy based on the relation induced by the strict inclusion of multisets, which facilitates the synthesis of arbitrary recursion structures, without having to indicate the recursion schemes in advance. The necessary auxiliary algorithms (like, e.g., for insertion and merging) are generated by the same principles from the synthesis conjectures that are automatically produced during the main proof, using a "cascading" method, which in fact contributes to the automation of theory exploration. The prover is implemented in the frame of the *Theorema* system and works in natural style, while the generated algorithms can be immediately tested in the same system.

Keywords: Deductive synthesis · Sorting · Lists · Binary trees · Multisets · Noetherian induction

1 Introduction

Automatic synthesis of algorithms is an interesting and challenging problem in automated reasoning, because algorithm invention appears to be difficult even

for the human intellect. Synthesis of sorting algorithms is especially challenging because the content and structure of the specification appears to be completely different from the expression of the algorithms. Thus case studies and automation attempts for synthesis of sorting algorithms have the potential of increasing our knowledge about possible general methods for algorithm synthesis.

We address the automated synthesis of algorithms which satisfy certain given specifications[3]. The specification is transformed into a synthesis conjecture from whose constructive proof a main algorithm is extracted. Usually this main algorithm needs some auxiliary algorithms, whose synthesis conjectures are produced during the main proof and then additional synthesis proofs are performed – the process may repeat as by "cascading" [4]. Our focus is on automating proofs for such conjectures, on the mechanical generation of the synthesis conjectures for the necessary auxiliary algorithms, and on the automatic extraction of the algorithms from the proofs. Cascading also constitutes a contribution to the automation of *theory exploration*[4] [3].

The implementation of the synthesis methods constitutes the automated proof–based synthesizer ***AlCons*** for sorting algorithms on lists and on binary trees using multisets, built as a prover in the *Theorema* system [7, 36] (based on *Mathematica* [5]). In order to illustrate the principles of the prover we present in this paper a summary of our experiments on binary trees. (Experiments on lists are presented in [16]).

The prover uses *general* inference rules and strategies for predicate logic, as well as *domain-specific* rules and strategies, which make the proof search more efficient.

1.1 Main contribution

The novelty of this work consists of: the use of *multisets* and the proof techniques related to them on binary trees, *nested use of cover sets*, the use of *cover sets on meta-variables*, the systematic principle for generating synthesis conjectures for the auxiliary functions (*cascading*), and the first description of the technical *implementation* in the current version of *Theorema*.

Multisets allow a very natural expression of the fact that two lists (trees) have the same objects[6]. More importantly, the use of multisets triggers some new proof techniques which make the proof search more efficient. Crucially for our current approach, we can use the Noetherian ordering on the domains of lists and of trees induced by the strict inclusion of the corresponding multisets, which is conveniently reflected at object level by the strict inclusion of multisets

[3] This ensures the correctness of algorithms and it is dual to algorithm verification, where the algorithms are first created and then checked.

[4] Theory exploration is the generation of interesting statements following from a certain set of axioms and/or for the purpose of developing certain proofs or algorithms.

[5] https://www.wolfram.com/mathematica

[6] In other approaches one uses the *permutation* notion, which must be expressed by specific algorithmic definitions, and whose properties are more difficult to infer.

of constants and variables occurring in the list/tree terms, and this allows a dynamic creation of concrete induction hypotheses according to the needs of the proof. Both lists and binary trees are addressed by ***AlCons*** with the same proof techniques[7], which demonstrates the possibility of generalization of such techniques to new domains, and also allows future work on algorithms combining lists and trees.

Cover sets and dynamic induction. Induction is implicitly realized using *cover sets*. A cover set [34] for a certain domain is a set of possibly non-ground terms whose set of ground instances covers the whole domain. From the algorithmic point of view the cover set represents a recipe for decomposing the input in order to be processed (technically it is applied to a certain Skolem constant), thus the synthesis will produce an equality (rewrite rule) for every cover set term – therefore we use mutually exclusive[8] terms. Every term from the cover set is used to generate an *induction conclusion* over a certain ground term (the *target object*). During the proof of this induction conclusion, the necessary *induction hypotheses* are generated dynamically by instantiating the induction conclusion with terms representing domain objects which are smaller in the Noetherian ordering than the current target object. Nested use of cover sets is novel w. r. t. the simple use of cover sets for realizing induction: while this technique allows to discover concrete induction principles by generating appropriate induction hypotheses during the proof, nested use of cover sets allows the discovery of nested recursions, which is rarely present in synthesized algorithms.

We extend the use of cover sets to meta–variables[9] in a similar way. Cover sets on meta–variables implement the algorithmic idea of combining intermediate results according to a certain "recipe" (because meta–variables represent the output of the computation). This is complementary to the use of cover–sets for Skolem constants, which implement the algorithmic idea of decomposing the input in a certain way (because the Skolem constants represent the input). In this way algorithmic ideas can be represented by proof techniques.

Cascading. Using specific heuristics, the prover decides when the current goal should be used for the creation of a conjecture for the synthesis of one or more auxiliary algorithms. This is proven separately and leads to the synthesis of one or more algorithms in the general context of the theory of lists/trees, thus it discovers some possibly interesting functions (therefore it contributes to the automation of theory exploration). The process can repeat in the new proofs, leading to more new functions. The synthesis conjectures for the auxiliary algorithms are generated by a novel strategy which: detects the need of an auxiliary function, produces the conjecture itself using the current proof situation, adds the appropriate property which allows new auxiliary function to be used later

[7] However some fine tuning of the implementation has been necessary, since trees have a more complex structure.

[8] Every element of the inductive domain is a ground instance of exactly one term from the cover set.

[9] Meta–variables designate terms (witnesses for existential goals) which are unknown at the current stage of the proof.

in the proof, and changes the current goal by inserting the new function calls at the appropriate places.

Implementation. The prover is implemented in the frame of the *Theorema* system, which offers a flexible and intuitive user interface, construction of theories and development of proofs in natural style, as well as direct execution of the synthesized algorithms, as they are produced in form of a set of (conditional) equalities. The implementation principles in the frame of *Theorema* 2.0 are in many respects different from the implementation in the previous version of *Theorema* and in the same time more powerful both from the point of view of interface as well as performance and readability. The prover is appropriate for the synthesis of sorting algorithms both for lists as well as for binary trees, however in this paper we describe only experiments on binary trees. An extensive presentation of experiments on lists is presented in [16].

1.2 Related work and originality

The problem of algorithms synthesis, including the synthesis of sorting algorithms is well-studied in the literature, but full automation of synthesis constitutes still a challenge. An overview of the most common approaches used to tackle the synthesis problem is given in [33]. Most approaches are based on special techniques for transformation of expressions (for program instance program transformation, Hoare-like or tableaux-like calculae). In contrast our approach emphasizes proving in natural style, and intuitive inference rules. Most synthesis methods use certain algorithm templates, or explicit induction schemas, while we use cover sets and dynamic induction instead. No other approaches use multisets, and only few address a systematic method for generating auxiliary algorithms.

Significant work has been done in the synthesis of sorting algorithms[10]. Six versions of sorting algorithms are derived in [10] by applying transformation rules. An extension of this work is in [1], see also [21]. Some specific transformation techniques which complement the ones in [10] are used by [19]. [26] classifies sorting algorithms.

[27] introduces deductive techniques for constructing recursive algorithms. [35] applies manually the techniques in [27] and derives several sorting algorithms in the theories of integers and strings. Later implementations using some of these principles are in [32, 24]. We follow some of the principles from [27, 35].

Systematic methods for generating auxiliary algorithms are also presented in [29, 27]. We use a different cascading strategy which transforms the failing goal together with the current assumptions into a new conjecture. [25] applies deductive tableau techniques [27], uses some heuristics and rippling [8] for the automated synthesis of several functions in Lisp in the theory of integers and lists. [25] shows how to prepare induction hypotheses to be used in the rippling proofs by using deductive rules.

[30] implemented the tool Synquid which is able to automatically synthesize several recursive algorithms operating on lists (including sorting algorithms)

[10] We presented a more detailed survey of the synthesis methods in [11].

and operations on trees (but not sorting), except the automatic synthesis of auxiliary functions. This work was extended by [20],using a technique based on some given templates in order to synthesize algorithms on lists and binary trees (e.g., converting a binary tree to a list, or a list to a binary search tree) together with some auxiliary functions. In [22] the authors describe an approach that combines deductive synthesis with cyclic proofs for automatically synthesizing recursive algorithms with recursive auxiliary functions and mutual recursion. They implement the tool Cypress and they synthesize algorithms operating on lists (including some sorting algorithms), and on trees (e.g., flattening a tree into a list, insertion, deletion, etc.). Their approach complements the one in [20] by considering a proof-driven approach instead of template-driven approach for synthesizing auxiliary functions. However, the synthesis of sorting algorithms on trees is not approached.

A valuable formalization, in a previous version of *Theorema* [6], of the synthesis of sorting algorithms is in [5], where an algorithm scheme is given together with the specification of the desired function. In contrast, ***AlCons*** uses cover–set decomposition and no algorithm scheme.

The theory of *multisets* (also called *bags*) is well studied in the literature, including computational formalizations, see e. g. [28]. The theory of multisets and a detailed survey of the literature related to multisets and their usage is presented in [2] and some interesting practical developments are in [31].

A systematic formalization of the theory of lists using multisets the correctness proofs of various sorting algorithms is mechanized in Isabelle/HOL[11], but it does not address algorithm synthesis. The use of multisets and of the special techniques related to them, as well as the systematic approach to the generation of synthesis conjectures for the auxiliary algorithms and the use of cover set induction constitute also significant improvements w. r. t. our previous work on this problem [11, 17].

Most related to this paper are our recent case studies on sorting algorithms for lists [13, 16, 14, 15], and some of the auxiliary algorithms on binary trees [12, 18]. The current paper presents the main principles and techniques resulting from these case studies, integrating all methods and improving them in order to realize a comprehensive tool, which works for both lists and binary trees. Also [12, 18] complement the current presentation with illustrative fragments of synthesis proofs of the auxiliary algorithms on binary trees. [12] applies explicit induction in order to derive some auxiliary algorithms, [18] extends [12] by applying cover sets and dynamic induction instead of explicit induction, and some different proof techniques for deriving several more auxiliary algorithms on binary trees. Complementary, the current paper presents the synthesis of several versions of sorting algorithms on binary trees and the synthesis of two more auxiliary algorithms (*SmallerEq*, *Bigger*), including *Insert* by using cover sets instead of explicit induction, and refines the proof techniques.

A prover for algorithm synthesis on lists [11] and another one for binary trees [17] was implemented in a previous version of *Theorema* [6]. There we

[11] https://isabelle.in.tum.de/library/HOL/HOL-Library/Sorting_Algorithms.html

use different synthesis methods and we do not use multisets. The novel system ***AlCons*** works both on lists and on binary trees.

Except for the previous work using the *Theorema* system, a distinctive feature of our approach is the use of natural style proving, and except for our own previous work, there is no approach in the literature to the direct sorting of binary trees.

2 Algorithm synthesis

2.1 Context and notation

Terms and formulae. Brackets are used for function and predicate application (like $f[x]$, $P[a]$). Quantifiers are denoted like $\underset{X}{\forall}$ and $\underset{Y}{\exists}$. Metavariables are starred (e.g., T^*, T_1^*, Z^*) and Skolem constants have integer indices (e.g., X_0, X_1, a_0).

Objects and theories. We consider three types: simple objects (*elements*) and composite objects (finite *binary trees* and finite *multisets*). Both in this presentation and in the prover typing is implicit, based on the notation conventions specified below.

Elements (denoted by a, b, c) are objects from a total ordered domain. The ordering on elements (notation $\leq$ and $<$) is extended to orderings between an element and a composite object and between composite objects, by requiring that all elements of the composite object observe the ordering relation[12].

Binary trees (denoted by L, R, S, T) are objects from an inductive domain: either ε (empty) or a triplet $\langle L, a, R\rangle$, where L and R are the left and right subtrees, and a is the root element.

Multisets (denoted by A, B, C) are objects whose elements can occur repeatedly. $\emptyset$ is the empty multiset, $\{\{a\}\}$ denotes the multiset containing the element a with multiplicity 1, and $\mathcal{M}[T]$ denotes the multiset of elements of a binary tree T. The union of multisets is additive $\uplus$ like in [23]. Some inference rules use implicitly the properties of union (commutativity, associativity, and unit $\emptyset$).

Knowledge. This contains the main properties of union of multisets, the definition of multisets of a tree, etc. For illustration the definition of sorted trees is:

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

2.2 Approach

The **specification** consists in an input condition $I[X, X', \ldots]$ applied to the inputs and an output condition $O[Y, X, X', \ldots]$ applied to the output Y and the same inputs. For the sorting problem the input condition is *True* (thus it is missing), but it may be present in the specification of some auxiliary algorithms. The output condition for sorting is: $\mathcal{M}[Y] = \mathcal{M}[X] \wedge \mathit{IsSorted}[Y]$, and

[12] Note that this introduces certain exceptions to antisymmetry and transitivity when the empty composite object is involved.

for the auxiliary functions is similar, but it typically contains some additional requirements. The **conjecture** corresponding to the specification is

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

In some experiments we use a conjecture of the form:

$$\underset{X,X',\ldots}{\forall}(I[X,X',\ldots] \implies O[F[X,X',\ldots],X,X',\ldots]),$$

where F is the name of the function to be synthesized.

The **proof** is developed by applying the techniques (inference rules and strategies) described in the sequel, and it generates one or more algorithms and possibly some conjectures for further synthesis (cascading).

The **algorithm** for a function $F[X,X',\ldots]$ is presented as a set of conditional equalities of the form:

$$Q[Y,Y',\ldots] \Rightarrow F[P[Y,Y',\ldots],P'[Y,Y',\ldots],\ldots] = T[Y,Y',\ldots],$$

where $P[Y,Y',\ldots],P'[Y,Y',\ldots],\ldots$ are patterns[13], Q is a formula, and T is a term. These conditional equalities can be applied as rewrite rules in order to compute F.

The theoretical basis and the correctness of this proof based synthesis scheme is well–known, see [27, 9] and was used in some recent publications by [11, 17], see also [13, 12, 16].

3 Proof Techniques

By proof techniques we understand *inference rules*, which describe one step of the proof, and *strategies*, which describe how to group several inference rules.

AlCons uses some of the common natural style inference rules, which are already implemented in *Theorema*: split assumed conjunction, Skolemization of the universal goal (but not of the existential assumptions), meta–variable for the existential goal (but not of the universal assumptions), rewriting by equality, matching and instantiation for forward and backward inferences, etc.

Some of the inference rules and strategies were first introduced in [13, 12, 16], and there we illustrate them on concrete examples on sorting and auxiliary algorithms on lists and on auxiliary algorithms on binary trees, however here they are first comprehensively integrated in one system and applied to synthesize sorting algorithms on binary trees.

We describe in the sequel only those techniques which are specific to ***AlCons*** and are very important for synthesis on binary trees.

[13] In our context, a pattern is a term possibly containing variables, whose ground instantiations define an injective function into the domain.

3.1 Inference Rules

IR-1: *Reduce composite argument.* Transform an atom of a goal (which is typically a conjunction of atoms) or an assumption (when it is an atom) into simpler atoms whose arguments do not contain function symbols. For the goal generate possibly few atoms, for the assumptions possibly many, because then some of the assumed atoms will match and cancel some of the goal atoms.
Example 1: $a \leq \mathit{Concat}[L_0, R_0]$ becomes $a \leq L_0 \wedge a \leq R_0$.
Example 2: $\mathit{IsSorted}[\langle T_1, a, T_2\rangle]$ becomes $\mathit{IsSorted}[T_1] \wedge T_1 \leq a \wedge a \leq T_2 \wedge \mathit{IsSorted}[T_2]$.
IR-2: *Simple goal as condition.* When the target metavariable already has a solution and the goal (after all possible reductions) is ground and contains only constant time functions and predicates[14], then this goal is taken as a condition and with the current solution to the metavariable it becomes a clause of the synthesized algorithm (see the partial proof in Fig. 2).
IR-3: *Use equivalence.* The equivalence relation between composite objects induced by the equality of the corresponding multisets is used to rewrite parts of the goal (or of the assumptions) by replacing composite objects with equivalent ones, when they occur in equality atoms or in ordering atoms.
Example 1: The goal $\mathcal{M}[\langle \mathit{Sort}[T_1^*], a^*, \mathit{Sort}[T_2^*]\rangle] = \mathcal{M}[T^*] \wedge \mathit{Sort}[T_1^*] \leq a^* \wedge a^* \leq \mathit{Sort}[T_2^*]$ becomes $\mathcal{M}[\langle T_1^*, a^*, T_2^*\rangle] = \mathcal{M}[T^*] \wedge T_1^* \leq a^* \wedge a^* \leq T_2^*$.
Example 2: The goal is $b \leq S_1$ is transformed into $b \leq L_0 \wedge b \leq a \wedge b \leq R_0$ using the assumption: $\mathcal{M}[S_1] = \mathcal{M}[L_0] \uplus \{\{a\}\} \uplus \mathcal{M}[R_0]$.
IR-4: *Expand multiset.* This rule expands a multiset term in the goal into several multiset terms. This is useful because then different groupings can be performed.
Example: The goal $\mathcal{M}[T^*] = \mathcal{M}[\langle L_0, a, R_0\rangle] \uplus \mathcal{M}[S_0]$, becomes $\mathcal{M}[T^*] = \mathcal{M}[L_0] \uplus \{\{a\}\} \uplus \mathcal{M}[R_0] \uplus \mathcal{M}[S_0]$.
IR-5: *Compress multiset.* This rule is the dual of the previous one, and it typically applies when the arguments contain terms which correspond to the recursive calls of the desired function. Example: if a part of the goal is $\mathcal{M}[T^*] = \mathcal{M}[T_1] \uplus \{\{a\}\} \uplus \mathcal{M}[T_2] \uplus \ldots$, then on one alternative branch[15] this part becomes $\mathcal{M}[T^*] = \mathcal{M}[\langle T_1, a, T_2\rangle] \uplus \ldots$. By repeated application this rule one reaches the situation of **IR-6**, as described in **ST-4**.
IR-6: *Solve metavariable.* When a part of the goal is $\mathcal{M}[X^*] = \mathcal{M}[\mathcal{T}]$ for a ground term $\mathcal{T}$, obtain the substitution $\{X^* \to \mathcal{T}\}$ and continue the proof with the remaining goal. In order to ensure the soundness, the prover keeps track of the order in which Skolem constants and metavariables have been introduced, and allows the use in a solution for a metavariable only the Skolem constants which have been generated before that metavariable.
IR-7: *Forward inference.* This rule is applied in order to produce new assumptions. If a ground atomic assumption matches a part of another (typically universal) assumption, instantiate the later and replace in it the resulting copy of

[14] This is just a matter of efficiency, the goal could contain anything as long as the currently synthesized function is not involved.
[15] The rule generates proof alternatives for different groupings of the multiset terms.

the ground assumption by the constant *True*, then simplify truth constants to produce a new assumption.
IR-8: *Backward inference.* Transform the goal using some assumption or a specific logical principle. If a ground atomic assumption matches a part of a ground or existential goal, instantiate the goal and replace in it the resulting copy of the ground assumption by the constant *True*, then simplify truth constants to produce a new goal.

3.2 Strategies

ST-1: *Cover set.* This strategy organizes the structure of each synthesis conjecture proof and the extraction of the synthesized algorithm, as in fact implements the Noetherian induction based on the ordering between objects induced by strict inclusion of multisets.

Each conjecture for the synthesis of a *target function* is a quantified statement over some *main universal variable*. A *cover set* is a set of universal terms[16] which represent the domain of the main universal variable, as described in [17].

We project this concept on Skolem constants: first the main universal variable is Skolemized ("arbitrary but fixed") — we call this the *target constant*, and we call the corresponding Skolemized goal the *target goal* – and then the corresponding cover–set terms are also grounded by Skolemization, we call these the *cover-set terms* and the corresponding constants the *cover-set constants*. The proof starts with a certain cover set (typically the one suggested by the recursive definition of the domain), and starts a proof branch for each ground term ("proof by cases"). On each proof branch the input conditions of the function are assumed, and then the existential variable corresponding to the output value of the function is transformed into a metavariable whose value (the "witness") will be found on the respective branch of the proof. Finally the algorithm will be generated as a set of [conditional] equalities: the terms of the cover set become arguments ("patterns") on the LHS of the equalities, and the corresponding witnesses become the RHS of these, after replacing back the Skolem constants by variables. The strategy can be applied in a *nested* way, by choosing a new target constant among the Skolem constants of the goal. Using this nesting scheme one can synthesize algorithms with nested recursion (see, e. g., **Algorithm 9**) as well as with recursion on several arguments, as for instance in the case of merging of lists in the merge-sort algorithm for lists (see [16], Algorithm 15).

Furthermore we use cover sets in a novel way also on meta–variables: this generates a certain structure for the synthesized algorithm by imposing on the result the structure of the corresponding term of the cover set (see for instance **Algorithm 5** for sorting).
ST-2: *Dynamic Induction.* (described in more detail in [13]) is used to dynamically generate induction hypotheses during the proof. When a ground term t represents an object which is smaller than the target constant X_0 of the target

[16] Terms containing universally quantified variables, such that for every element of the domain there exists exactly one term in the set which instantiates to that element.

goal $P[X_0]$, then $P[t]$ is added as a new assumption, but modified by inserting the corresponding call of the target function instead of the existential variable.

This strategy is applied in a similar manner to metavariables, when they occur in the goal. When a metavariable Y^* represents an object which is smaller than the target constant X_0, then $P[Y^*]$ may be added as new assumption.

ST-3: *Cascading.* This strategy consists in proving separately a conjecture for synthesizing the algorithm for some auxiliary functions needed in the current proof. The Skolem constants from the current goal become universal variables $x, x', \ldots$, the metavariables from the current goal become existential variables $y, y', \ldots$, and the conjecture has the structure[17]:

$$\underset{x\,x'}{\forall\forall} \ldots (P[x, x', \ldots] \Longrightarrow \underset{y\,y'}{\exists\exists} \ldots Q[x, x', \ldots, y, y', \ldots]) \tag{1}$$

$P[x, x', \ldots]$ is composed from the assumptions which contain *only* the Skolem constants present in the goal, and $Q[x, x', \ldots, y, y', \ldots]$ is composed from the goal. A successful proof of the conjecture generates the functions $f[x, x', \ldots]$, $f'[x, x', \ldots], \ldots$, which have the property:

$$\underset{x\,x'}{\forall\forall} \ldots (P[x, x', \ldots] \Longrightarrow Q[x, x', \ldots, f[x, x', \ldots], f'[x, x', \ldots], \ldots]) \tag{2}$$

The current proof continues after adding this property to the assumptions[18], thus if some of the generated functions are necessary later in the proof, they can be used without a new cascading step. The new assumption will trigger the simplification of the current goal by inserting the auxiliary function.

ST-4: *Group multisets.* This strategy uses **IR-5** and applies when the goal contains an equality of the form: $\mathcal{M}[Y^*] = \mathcal{M}[t_1] \uplus \mathcal{M}[t_2] \uplus \ldots$, where Y^* is the metavariable we need to solve, and $t_1, t_2, \ldots$ are ground terms. The flow of the proof consists in transforming the union on the RHS of the equality into a single $\mathcal{M}[t]$, because this gives the solution $Y^* \rightarrow t$. The prover groups pairs or triplets of operands of $\uplus$ together (no matter whether they are contingent or not, because commutativity) and creates an alternative for each group. On each alternative the multiset term which equals the union of the group is constructed by application of the appropriate function in one of the following ways:

1. the auxiliary function is already known, the proof works by predicate logic;
2. induction can be applied (if the target function has the same structure);
3. a separate synthesis proof of the function is necessary by **ST-3** (cascading).

3.3 Implementation

In the *Theorema* system, the proof develops as a tree of *proof situations*, each consisting of a set of assumptions and a goal, and also other various information

[17] By local convention, here x, x', y, y' represent any kind of objects.

[18] Note that these kind of new assumptions are *global*: they can be used on any branch of the current proof.

which may be prover specific. Every proof situation is transformed into one or more proof situations by applying an *inference rule*, which creates or modifies the goal and/or one or more assumptions, and thus extends the proof tree. When several proof situations are created, there are two types of proof tree nodes: the AND nodes (all subproofs must succeed), and the OR nodes (at least one subproof must succeed – these are "proof alternatives"). Many inference rules produce alternatives (e. g. *compress multiset, backward chaining*), from which some may be unsuccessful. Each successful alternative has typically several AND branches, each of them corresponding to a clause in the definition of the synthesized algorithm. Since the cover set strategy is applied in a nested way, the proof tree is theoretically infinite, and may produce an arbitrary number of algorithms. The concrete proofs are however finite because we limit the depth of the proof tree.

As it is usual in the *Theorema* system, our prover consists of a collection of rewrite rules which correspond to the intended inferences. Each rule rewrites the *proof situation* into new one, and produces additionally a *proof information* (a list of elements necessary for the presentation of the proof). The proof information is language independent and is aggregated in a tree wich represents finally the whole proof: the *proof object.* Using a set of language-dependent rewrite rules corresponding to the proof steps, the proof object is finally transformed in a *Mathematica* notebook explaining the proof.

In our case the proof object also contains the information relevant for the synthesized algorithm, which is extracted automatically at the end of the proof.

Contextual Information. Besides the current goal and the list of the current assumptions, which are the core elements of the proof situation, the prover uses certain contextual information for guiding the realization of the inference rules and strategies. The contextual information is split into *global* and *local.*

The **global context** consists of constants which are available to the prover on all branches of the proof. This contains among other: the table with the names of the variables assigned to the different types, the table with the cover sets corresponding to lists and to trees, and the list of rewrite rules for the simplification of truth constants.

The **local context** consists of information which is specific to every branch of the proof, and it is dynamically updated. This contains: the type table, which indicates the type of each item; the target goal, the target Skolem constant and the target meta-variable used for the realization of the strategy **ST-1** (cover set); the table of the Noetherian relation between Skolem constants, used for induction; and the table of rewrite rules corresponding to the current assumptions (see below), etc.

In order to ease the use of the current assumptions, they are reflected in certain rewrite rules in a special table of the local context. When new goals [assumptions] are produced, the prover tries to simplify them using these rules, depending on the situation in the proof.

The tables composing the local context are implemented as associative memory structures: each element or group of elements is associated with a textual

keyword. This makes it easy to access an element by using the rewrite mechanism provided by *Mathematica*, and also to write inference rules based on pattern matching.

Both the global and the local contexts are implemented in a generic way, both the structures and the manipulating functions are type independent, thus any relevant information (like, e. g., cover set information) can be added to the context and maintained by the functions provided, without having to change the implementation.

The *Theorema* system[19] and an example of the prover usage[20] are available online.

4 Experiments on binary trees

In order to illustrate the proof techniques of ***AlCons*** we summarize in this section our experiments on binary trees. (The experiments on lists are detailed in [16].) The synthesized algorithms relevant to binary trees are:

(i) sorting algorithms (not yet presented in our papers): **Algorithm 1** (which uses *Insert*, *Concat*), **Algorithm 2**, **Algorithm 3**, **Algorithm 4** (which use *Insert*, *Merge*), **Algorithm 5** (which uses *Concat*, *SmallerEq*, *Bigger*), **Algorithm 6** (which uses *Merge*, *SmallerEq*, *Bigger*), as well as some similar versions of them;

(ii) auxiliary algorithms: *Insert* (**Algorithm 7**) [12], derived here by different techniques; numerous versions of *Concat*: the synthesis of **Algorithm 8** and first three similar versions of it are presented in [12], and other twenty versions in [18]; four versions of *Merge*: the synthesis of the first two (**Algorithm 9**, **Algorithm 10**) is presented in [12], and the other two in [18]; novel: *SmallerEq* (**Algorithm 11**) and *Bigger* (**Algorithm 12**).

The algorithms presented also in [12] are generated using explicit induction (thus the user has to anticipate the structure of the algorithm), and the algorithms 1 to 4 can also be derived in this way. In contrast, by using cover sets and dynamic induction, all algorithms mentioned above are synthesized without any prior anticipation of the algorithm structure. Moreover, algorithms 5 and 6 (and their similar versions), as well as the selection auxiliary functions *SmallerEq* and *Bigger* are consequent to the use of the novel paradigm of applying cover sets to meta-variables.

The following subsections illustrate the process of synthesis by describing some parts of the proofs.

4.1 Sorting algorithms.

The synthesis conjecture is:

[19] `https://www.risc.jku.at/research/theorema/software/`
[20] `https://www.risc.jku.at/people/tjebelea/AlCons.html`

Conjecture 1. $\underset{XT}{\forall\exists}(\mathcal{M}[T] = \mathcal{M}[X] \wedge \mathit{IsSorted}[T])$.

The goal after Skolemization and introduction of the meta–variable is:

$$\mathcal{M}[T^*] = \mathcal{M}[X_0] \wedge \mathit{IsSorted}[T^*]. \tag{3}$$

Strategy **ST-1** (cover set) starts two branches: on the Skolem constant and on the meta-variable.
Branch 1. Strategy **ST-1** applies to X_0 using the cover set $\{\varepsilon, \langle L_0, a_0, R_0\rangle\}$ and generates two cases:
Case 1.1: $X_0 = \varepsilon$ is trivial and the solution is $\{T^* \to \varepsilon\}$.
Case 1.2: $X_0 = \langle L_0, a_0, R_0\rangle$. The goal becomes:

$$\mathcal{M}[T^*] = \mathcal{M}[\langle L_0, a_0, R_0\rangle] \wedge \mathit{IsSorted}[T^*]. \tag{4}$$

This is expanded by **IR-4** (expand multiset) into:

$$\mathcal{M}[T^*] = \mathcal{M}[L_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[R_0] \wedge \mathit{IsSorted}[T^*]. \tag{5}$$

Strategy **ST-4** (pair multisets) applies on goal (5) and then strategy **ST-3** (cascading) generates the conjectures corresponding to the synthesis of *Concat*, and *Merge* (see details in [12]) on two different cases, adds the assumptions expressing the properties of these auxiliary functions, and rewrites the goal in each case by using *Concat* and *Merge*, respectively.
Case 1.2.1 Goal (5) becomes:

$$\mathcal{M}[T^*] = \{\{a_0\}\} \uplus \mathcal{M}[\mathit{Concat}[L_0, R_0]] \wedge \mathit{IsSorted}[T^*]. \tag{6}$$

Strategy **ST-2** (induction) uses $\mathit{Concat}[L_0, R_0]$, which is smaller in the Noetherian ordering than $\langle L_0, a_0, R_0\rangle$, to produce the assumption:

$$\begin{gathered}\mathcal{M}[\mathit{Sort}[\mathit{Concat}[L_0, R_0]]] = \mathcal{M}[\mathit{Concat}[L_0, R_0]] \wedge \\ \mathit{IsSorted}[\mathit{Sort}[\mathit{Concat}[L_0, R_0]]].\end{gathered} \tag{7}$$

Goal (5) is rewritten using (7) into:

$$\mathcal{M}[T^*] = \{\{a_0\}\} \uplus \mathcal{M}[\mathit{Sort}[\mathit{Concat}[L_0, R_0]]] \wedge \mathit{IsSorted}[T^*]. \tag{8}$$

ST-4 applied to $\{\{a_0\}\}$ and $\mathcal{M}[\mathit{Sort}[\mathit{Concat}[L_0, R_0]]]$ uses **ST-3** to produce *Conjecture 2* for the synthesis of *Insert*. By s **ST-3** the generated assumption is:

$$\begin{gathered}\underset{X}{\forall}\Big(\mathit{IsSorted}[X] \Longrightarrow \\ \underset{a}{\forall}\big(\mathcal{M}[\mathit{Insert}[a, X]] = \{\{a\}\} \uplus \mathcal{M}[X] \ \wedge \mathit{IsSorted}[\mathit{Insert}[a, X]]\big)\Big).\end{gathered} \tag{9}$$

and goal (5) becomes

$$\mathcal{M}[T^*] = \mathcal{M}[\mathit{Insert}[a_0, \mathit{Sort}[\mathit{Concat}[L_0, R_0]]]] \wedge \mathit{IsSorted}[T^*]. \tag{10}$$

The solution for T^* is $\mathit{Insert}[a_0, \mathit{Sort}[\mathit{Concat}[L_0, R_0]]]$. The proof succeeds on this branch and the extracted algorithm is:

Algorithm 1 Sorting trees, version 1.

$$\underset{a,L,R}{\forall}\left(\begin{array}{c} Sort[\varepsilon] = \varepsilon \\ Sort[\langle L, a, R\rangle] = Insert[a, Sort[Concat[L, R]]] \end{array}\right)$$

Case 1.2.2 Goal (5) becomes:

$$\mathcal{M}[T^*] = \{\{a_0\}\} \uplus \mathcal{M}[Merge[L_0, R_0]] \ \wedge IsSorted[T^*]. \tag{11}$$

Strategy **ST-2** uses $Merge[L_0, R_0]$ (which is smaller than $\langle L_0, a_0, R_0\rangle$) to produce the assumption:

$$\begin{array}{c} \mathcal{M}[Merge[Sort[L_0], Sort[R_0]]] = \mathcal{M}[Merge[L_0, R_0]] \wedge \\ IsSorted[Merge[Sort[L_0], Sort[R_0]]]. \end{array} \tag{12}$$

Goal (5) is rewritten using (12) into:

$$\mathcal{M}[T^*] = \{\{a_0\}\} \uplus \mathcal{M}[Merge[Sort[L_0], Sort[R_0]]] \wedge IsSorted[T^*]. \tag{13}$$

Strategy **ST-4** applied to $\{\{a_0\}\}$ and $\mathcal{M}[Merge[Sort[L_0], Sort[R_0]]]$ uses now the already known function *Insert* to update the goal into:

$$\mathcal{M}[T^*] = Insert[a_0, Merge[Sort[L_0], Sort[R_0]]] \wedge IsSorted[T^*]. \tag{14}$$

This gives a solution for T^* and the algorithm:

Algorithm 2 Sorting trees, version 2. $Insert[a, Merge[Sort[L], Sort[R]]]$

$$\underset{a,L,R}{\forall}\left(\begin{array}{c} Sort[\varepsilon] = \varepsilon \\ Sort[\langle L, a, R\rangle] = Insert[a, Merge[Sort[L], Sort[R]]] \end{array}\right)$$

Remark: Since for all sorting algorithms the base case is the same, as well as the LHS of the recursive equality, we to state only its RHS for the other algorithms.

The proof is similar for two other cases produced by **ST-4** from goal (5) by grouping first the unit multiset with another, and generates:

Algorithm 3 Sorting trees, version 3. $Merge[Sort[L], Insert[a, Sort[R]]]$

Algorithm 4 Sorting trees, version 4. $Merge[Insert[a, Sort[L]], Sort[R]]$

Branch 2. **ST-1** applies to T^* using the cover set $\{\varepsilon, \langle L^*, a^*, R^*\rangle\}$ and two cases are generated:
Case 2.1: $T^* = \varepsilon$ is trivial.
Case 2.2: $T^* = \langle L^*, a^*, R^*\rangle$. The goal becomes:

$$\mathcal{M}[\langle L^*, a^*, R^*\rangle] = \mathcal{M}[\langle L_0, a_0, R_0\rangle] \wedge IsSorted[T^*]. \tag{15}$$

This is transformed by **IR-4** (expand multiset) and **IR-1** (reduce composite argument on *IsSorted*) into:

$$\begin{array}{c} \mathcal{M}[L^*] \uplus \{\{a^*\}\} \uplus \mathcal{M}[R^*] = \mathcal{M}[L_0] \uplus \{\{a_0\}\} \uplus \mathcal{M}[R_0] \wedge \\ IsSorted[L^*] \ \wedge IsSorted[R^*] \ \wedge L^* \leq a^* < R^*. \end{array} \tag{16}$$

Using the equality the prover computes the partial solution $a^* = a_0$ and reduces the goal correspondingly, and then **ST-4** starts two alternatives:
Case 2.2.1 By pairing $\mathcal{M}[L_0], \mathcal{M}[R_0]$ using *Concat* the goal becomes:

$$\begin{array}{c} \mathcal{M}[L^*] \uplus \mathcal{M}[R^*] = \mathcal{M}[Concat[L_0, R_0]] \wedge \\ IsSorted[L^*] \ \wedge IsSorted[R^*] \ \wedge L^* \leq a_0 < R^*. \end{array} \tag{17}$$

By **ST-2** (dynamic induction) $Concat[L_0, R_0]$ is replaced by $Sort[Concat[L_0, R_0]]$, and then **ST-3** generates *Conjecture 6* for the synthesis of *SmallerEq* and *Bigger*, adds the corresponding properties of them to the global assumptions, and updates the goal to:

$$\begin{array}{c} \mathcal{M}[L^*] \uplus \mathcal{M}[R^*] = \\ \mathcal{M}[SmallerEq[a_0, Sort[Concat[L_0, R_0]]]] \uplus \mathcal{M}[Bigger[a_0, Sort[Concat[L_0, R_0]]]] \wedge \\ IsSorted[L^*] \ \wedge IsSorted[R^*] \ \wedge L^* \leq a_0 < R^*. \end{array} \tag{18}$$

This gives the obvious solutions to L^*, R^* and the algorithm:

Algorithm 5 Sorting trees, version 5.

$$\underset{a,L,R}{\forall} \left(\begin{array}{c} Sort[\langle L, a, R\rangle] = \\ \langle SmallerEq[a, Sort[Concat[L, R]]], a, Bigger[a, Sort[Concat[L, R]]]\rangle \end{array} \right)$$

(In an efficient implementation $Sort[Concat[L, R]]$ must be computed only once.)
Case 2.2.1 In a similar way but with different pairing of multiset terms, and using the already known selection functions, one obtains the algorithm:

Algorithm 6 Sorting trees, version 6.

$$\underset{a,L,R}{\forall} \left(\begin{array}{c} Sort[\langle L, a, R\rangle] = \\ \langle Merge[SmallerEq[a, Sort[L]], SmallerEq[a, Sort[R]]]], \\ a, \\ Merge[Bigger[a, Sort[L]], Bigger[a, Sort[R]]]]\rangle \end{array} \right)$$

Several similar versions of the latest two algorithms are generated by ST-4 permuting the multiset terms corresponding to L and R.

4.2 Auxiliary algorithms.

Insert. Inserts an element in a sorted tree such that the result remains sorted.

Conjecture 2. $\underset{a}{\forall}\underset{X}{\forall}\Big(IsSorted[X] \Longrightarrow \underset{S}{\exists}\Big(\mathcal{M}[S] = \{\{a\}\} \uplus \mathcal{M}[X] \wedge IsSorted[S]\Big)\Big)$

is used in the practical experiment as:

Conjecture 3.
$\underset{X}{\forall}\Big(IsSorted[X] \Longrightarrow \underset{a}{\forall}\Big(\mathcal{M}[Insert[a, X]] = \{\{a\}\} \uplus \mathcal{M}[X] \wedge IsSorted[Insert[a, X]]\Big)\Big)$

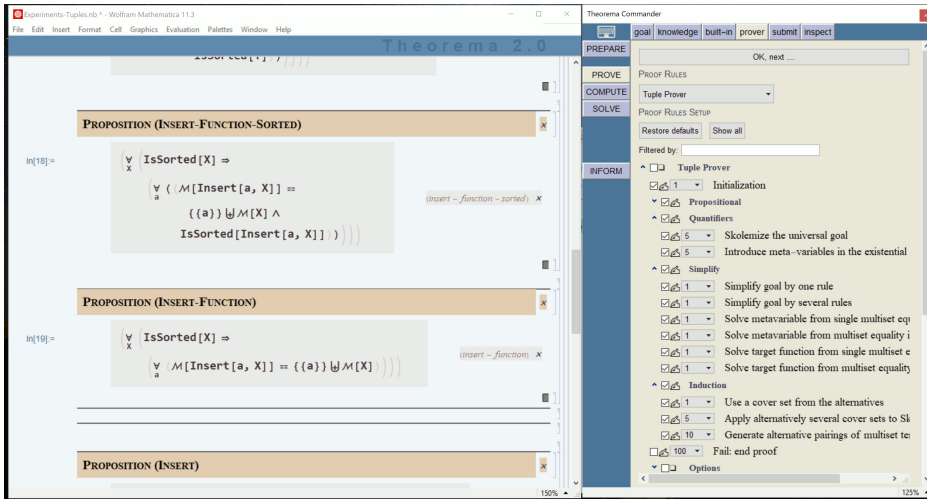

Fig. 1. Setup for proving *Conjecture 3*.

Fig. 1 shows the formalization of the conjecture in *Theorema* and the graphical user interface of the prover.

The proof uses the cover set $\{\varepsilon, \langle L_0, b_0, R_0\rangle\}$ for the Skolem constant X_0 and generates the algorithm:

Algorithm 7 Insertion in a sorted tree.

$$\forall_{a,b,L,R} \left(\begin{array}{c} Insert[a, \varepsilon] = \langle \varepsilon, a, \varepsilon \rangle \\ Insert[a, \langle L, b, R\rangle] = \left\{ \begin{array}{l} \langle Insert[a, L], b, R\rangle, \text{ if } a \le b \\ \langle L, b, Insert[a, R]\rangle, \text{ if } b < a \end{array} \right. \end{array} \right)$$

Figure 2 shows a part of the proof of the conjecture, with the successful generation of the first clause of the algorithm.

This algorithm was derived with different methods in [17] and by explicit induction in [12] instead of using cover sets.

Concat. Combine two [unsorted] trees into an [unsorted] tree.

Conjecture 4. $\forall_{X}\forall_{Y}\exists_{Z}\Big(\mathcal{M}[Z] = \mathcal{M}[X] \uplus \mathcal{M}[Y]\Big)$

From the proof of this conjecture 24 versions of *Concat* algorithm are extracted. The first 4 versions are also derived in [12] and the other 20 are in [18].

Algorithm 8 Concatenation of trees, version 1.

$$\forall_{a,L,R,S} \left(\begin{array}{c} Concat[\varepsilon, S] = S \\ Concat[\langle L, a, R\rangle, S] = \langle L, a, Concat[R, S]\rangle \end{array} \right)$$

The other generated versions are essentially the same but permute L, R, S and the two main branches of the resulting tree.

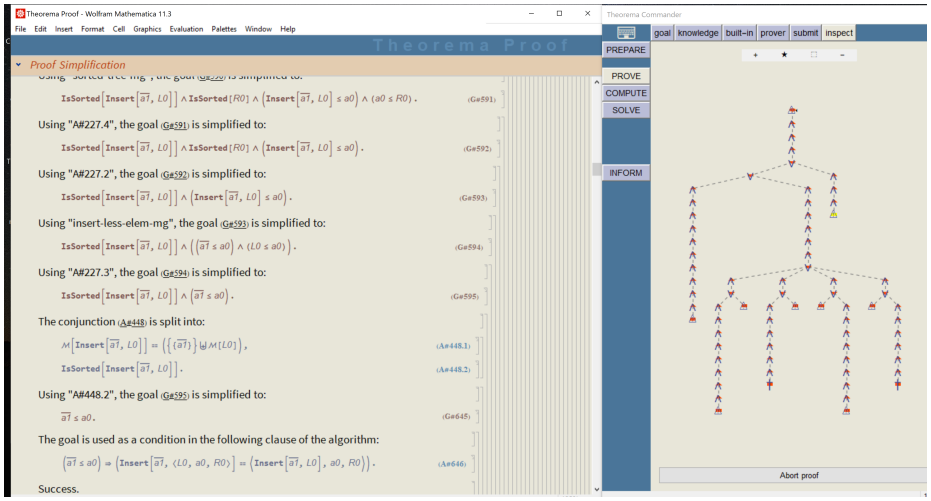

Fig. 2. Part of the generated proof of *Conjecture 3* and the proof tree.

Merge. Combine two sorted trees into a sorted tree.

Conjecture 5.
$$\underset{XY}{\forall\forall}\Big((IsSorted[X] \wedge IsSorted[Y]) \Longrightarrow \underset{Z}{\exists}\Big(\mathcal{M}[Z] = \mathcal{M}[X] \uplus \mathcal{M}[Y] \wedge IsSorted[Z]\Big)\Big)$$

From the proof of this the following two versions of *Merge* are extracted [12]:

Algorithm 9 Merge sorted trees, version 1.
$$\underset{a,L,R,S}{\forall}\left(\begin{array}{c} Merge[\varepsilon, S] = S \\ Merge[\langle L, a, R\rangle, S] = Merge[L, Merge[R, Insert[a, S]]] \end{array}\right)$$

Algorithm 10 Merge sorted trees, version 2 (the inductive step).
$$\underset{a,L,R,S}{\forall}\big(\, Merge[\langle L, a, R\rangle, S] = Merge[Concat[L, R], Insert[a, S]]\,\big)$$

as well as (by applying cover sets to meta-variables), the following two versions described in [18], which have in the second equality:
$\langle SmallerEq[a, Merge[L, S]], a, Bigger[a, Merge[R, S]]\rangle$,
$\langle Merge[L, SmallerEq[a, S]], a, Merge[R, Bigger[a, S]]\rangle$.

***SmallerEq* and *Bigger*.** Select from a sorted tree the elements that are smaller, respectively bigger than a given element.

Conjecture 6. $\underset{X}{\forall}\Big(IsSorted[X] \Longrightarrow \underset{a}{\forall}\underset{T_1}{\exists}\underset{T_2}{\exists}\Big(\mathcal{M}[X] = \mathcal{M}[T_1] \uplus \mathcal{M}[T_2] \quad \wedge$
$$T_1 \leq a \;\wedge a < T_2 \wedge\; IsSorted[T_1] \wedge IsSorted[T_2]\Big)\Big).$$

Algorithm 11

$$\underset{a,b,L,R}{\forall}\left(\begin{array}{c} SmallerEq[a,\varepsilon]=\varepsilon \\ SmallerEq[a,\langle L,b,R\rangle]=\begin{cases}\langle SmallerEq[a,L],a,SmallerEq[a,R]\rangle & \text{if } a=b\\ SmallerEq[a,L], & \text{if } a<b\\ \langle L,b,SmallerEq[a,R]\rangle, & \text{if } b<a\end{cases}\end{array}\right)$$

Algorithm 12

$$\underset{a,b,L,R}{\forall}\left(\begin{array}{c} Bigger[a,\varepsilon]=\varepsilon \\ Bigger[a,\langle L,b,R\rangle]=\begin{cases}\langle Bigger[a,L],b,R\rangle, & \text{if } a<b\\ Bigger[a,R], & \text{if } b\leq a\end{cases}\end{array}\right)$$

5 Conclusions and further work

This paper gives the description of ***AlCons***, a powerful system for proof–based algorithm synthesis on lists and binary trees using multisets. The proofs generated by ***AlCons*** are easy to understand (similar to human proofs) and they are generated in a few seconds.

The most important proof strategies are: use cover sets together with multiset based Noetherian induction, pairing of multisets, and cascading. By using cover sets, no algorithm scheme and no concrete induction principles are needed in advance, as they are dynamically produced during the proof, and even nested induction algorithms can be generated automatically.

As future work one can extend ***AlCons*** to generate algorithms which combine operations on both lists and trees (e.g., algorithms for transforming a tree in a sorted list, transforming a non–sorted list into a balanced binary search tree), as well as more complex algorithms for sorting and searching – for instance on balanced trees. Moreover one can extend the prover with capabilities for automatic analysis of time and space complexity of the synthesized algorithms.

References

1. Barstow, D.R.: Remarks on "a synthesis of several sorting algorithms" by John Darlington. Acta Informatica **13**, 225–227 (1980)
2. Blizard, W.D.: Multiset Theory. Notre Dame Journal of Formal Logic **30**(1), 36–66 (1989). https://doi.org/10.1305/ndjfl/1093634995
3. Buchberger, B.: Theory Exploration with Theorema. Analele Universitatii Din Timisoara, Seria Matematica-Informatica **XXXVIII**(2), 9–32 (2000)
4. Buchberger, B.: Algorithm Invention and Verification by Lazy Thinking. Analele Universitatii din Timisoara, Seria Matematica - Informatica **XLI**, 41–70 (2003)
5. Buchberger, B., Craciun, A.: Algorithm Synthesis by Lazy Thinking: Using Problem Schemes. In: Proceedings of SYNASC 2004. pp. 90–106 (2004)
6. Buchberger, B., Dupre, C., Jebelean, T., Kriftner, F., Nakagawa, K., Vasaru, D., Windsteiger, W.: The Theorema project: A progress report. In: Calculemus 2000. pp. 98–113. A.K. Peters, Natick, Massachusetts (2000)

7. Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., Windsteiger, W.: Theorema 2.0: Computer-Assisted Natural-Style Mathematics. Journal of Formalized Reasoning **9**(1), 149–185 (2016). https://doi.org/10.6092/issn.1972-5787/4568
8. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: meta-level guidance for mathematical reasoning. Cambridge University Press (2005)
9. Bundy, A., Dixon, L., Gow, J., Fleuriot, J.: Constructing Induction Rules for Deductive Synthesis Proofs. Electronic Notes Theoretical Computer Science **153**, 3–21 (March 2006). https://doi.org/10.1016/j.entcs.2005.08.003
10. Darlington, J.: A synthesis of several sorting algorithms. Acta Informatica **11**, 1–30 (1978)
11. Dramnesc, I., Jebelean, T.: Synthesis of List Algorithms by Mechanical Proving. Journal of Symbolic Computation **68**, 61–92 (2015). https://doi.org/10.1016/j.jsc.2014.09.030
12. Dramnesc, I., Jebelean, T.: Automatic synthesis of merging and inserting algorithms on binary trees using multisets in *theorema*. In: MACIS 2019. pp. 153–168. LNCS, Springer (2019)
13. Dramnesc, I., Jebelean, T.: Proof–Based Synthesis of Sorting Algorithms Using Multisets in *Theorema*. In: FROM 2019. pp. 76–91. EPTCS 303 (2019). https://doi.org/10.4204/EPTCS.303.6
14. Dramnesc, I., Jebelean, T.: Deductive synthesis of Bubble-Sort using multisets. In: SAMI 2020. pp. 165–172. IEEE (2020). https://doi.org/10.1109/SAMI48414.2020.9108725
15. Dramnesc, I., Jebelean, T.: Deductive synthesis of Min-Max-Sort using multisets. In: SACI 2020. pp. 165–172. IEEE (2020). https://doi.org/10.1109/SACI49304.2020.9118814
16. Dramnesc, I., Jebelean, T.: Synthesis of sorting algorithms using multisets in Theorema. Journal of Logical and Algebraic Methods in Programming **119**(100635) (2020). https://doi.org/10.1016/j.jlamp.2020.100635
17. Dramnesc, I., Jebelean, T., Stratulat, S.: Mechanical Synthesis of Sorting Algorithms for Binary Trees by Logic and Combinatorial Techniques. Journal of Symbolic Computation **90**, 3–41 (2019). https://doi.org/10.1016/j.jsc.2018.04.002
18. Dramnesc, I., Jebelean, T.: Synthesis of merging algorithms on binary trees using multisets in Theorema. In: SACI 2021. pp. 497–502. IEEE (2021). https://doi.org/10.1109/SACI51354.2021.9465619
19. Dromey, R.G.: Derivation of sorting algorithms from a specification. Computer Journal **30**(6), 512–518 (1987)
20. Eguchi, S., Kobayashi, N., Tsukada, T.: Automated synthesis of functional programs with auxiliary functions. In: APLAS 2018. pp. 223–241 (2018). https://doi.org/10.1007/978-3-030-02768-1_13
21. Howard, B.T.: Another iteration on "A synthesis of several sorting algorithms" Technical Report KSU CIS 94-8, Department of Computing and Information Sciences, Kansas State University (1994)
22. Itzhaky, S., Peleg, H., Polikarpova, N., Rowe, R.N.S., Sergey, I.: Cyclic program synthesis. In: PLDI '21. pp. 944–959. ACM (2021). https://doi.org/10.1145/3453483.3454087
23. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley, 3 edn. (1998). https://doi.org/10.1137/1012065
24. Korukhova, Y.: Automatic Deductive Synthesis of Lisp Programs in the System ALISA. In: JELIA 2006. pp. 242–252. Springer LNAI 4160 (2006)

25. Korukhova, Y.: An Approach to Automatic Deductive Synthesis of Functional Programs. Annals of Mathematics and Artificial Intelligence **50**(3-4), 255–271 (2007). https://doi.org/10.1007/s10472-007-9079-9
26. Lau, K.K.: Top-down synthesis of sorting algorithms. The Computer Journal **35**, A001–A007 (1992)
27. Manna, Z., Waldinger, R.: A Deductive Approach to Program Synthesis. ACM Transactions on Programming Languages and System **2**(1), 90–121 (1980). https://doi.org/10.1145/357084.357090
28. Manna, Z., Waldinger, R.: The Logical Basis for Computer Programming, vol. 1: Deductive Reasoning. Addison-Wesley (1985). https://doi.org/10.2307/2275898
29. Manna, Z., Waldinger, R.: Fundamentals Of Deductive Program Synthesis. IEEE Transactions on Software Engineering **18**(8), 674–704 (1992). https://doi.org/10.1109/32.153379
30. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: PLDI 2016. pp. 522–538 (2016). https://doi.org/10.1145/2908080.2908093
31. Radoaca, A.: Properties of Multisets Compared to Sets. In: SYNASC 2015. pp. 187–188 (2015). https://doi.org/10.1109/SYNASC.2015.37
32. Smith, D.R.: Kids: a semiautomatic program development system. IEEE Transactions on Software Engineering **16**(9), 1024–1043 (1990). https://doi.org/10.1109/32.578788
33. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. SIGPLAN Not. **45**(1), 313–326 (Jan 2010). https://doi.org/10.1145/1707801.1706337
34. Stratulat, S.: A general framework to build contextual cover set induction provers. J. of Symbolic Computation **32**, 403–445 (2001)
35. Traugott, J.: Deductive Synthesis of Sorting Programs. Journal of Symbolic Computation **7**(6), 533–572 (1989). https://doi.org/10.1016/S0747-7171(89)80040-9
36. Windsteiger, W.: Theorema 2.0: A System for Mathematical Theory Exploration. In: ICMS'2014. LNCS, vol. 8592, pp. 49–52 (2014). https://doi.org/10.1007/978-3-662-44199-2_9