Synthesis of Some Algorithms for Trees: Experiments in Theorema

Isabela Drămnesc[1], Tudor Jebelean[2] and Sorin Stratulat[3]

[1] Department of Computer Science
West University, Timisoara, ROMANIA
idramnesc@info.uvt.ro
[2] Research Institute for Symbolic Computation
Johannes Kepler University, Linz, AUSTRIA
Tudor.Jebelean@jku.at
[3] LITA, Department of Computer Science
Université de Lorraine, Metz, FRANCE
sorin.stratulat@univ-lorraine.fr

Abstract. We develop various proof techniques for the synthesis of sorting algorithms on binary trees, by extending our previous work on the synthesis of algorithms on lists. Appropriate induction principles are designed and various specific prove-solve methods are experimented, mixing rewriting with assumption-based forward reasoning and goal-based backward reasoning *à la* Prolog.
The proof techniques are implemented in the *Theorema* system and are used for the automatic synthesis of several algorithms for sorting and for the auxiliary functions, from which we present few here. Moreover we formalize and check some of the algorithms and some of the properties in the *Coq* system.

1 Introduction

Program synthesis is currently a very active area of programming language and verification communities. Generally speaking, the program synthesis problem consists in finding an algorithm which satisfies a given specification. We focus on the proof-based synthesis of functional algorithms, starting from their formal specification expressed as two predicates: the input condition $I[X]$ and the output condition $O[X,T]$. The desired function F must satisfy the correctness condition $(\forall X)(I[X] \Longrightarrow O[X,F[X]])$.[4]

We are interested to develop proof-based methods for finding F and to build formal tools for mechanizing and (partially) automatizing the proof process, by following constructive theorem proving and program extraction techniques to deductively synthesize F as a functional program [6]. The way the constructive proof is built is essential since the definition of F can be extracted as a side effect of the proof. 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 definitions of F. The extraction procedure guarantees that F satisfies the specification.

[4] The square brackets have been used for function and predicate applications instead of round brackets.

Non-trivial algorithms, as for sorting [16], are generated when X is a recursively-defined unbounded data structure, as lists and trees. In this paper, we apply the deductive approach to synthesize binary tree algorithms, extending similar results for lists [9–12]. In order to do this, we introduce new induction principles, proof strategies and inference rules based on properties of binary trees. Numerous new algorithms have been synthesized. The correctness of the discovered algorithms is ensured by the soundness of the induction principles, the specific inference rules and proof strategies introduced in this paper.

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[5] and e.g., [5] which is itself implemented in Mathematica [24]. *Theorema* offers significant support for automatizing 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 are presented in Section 5.

Additionally we have formalized part of the theory presented here and mechanically checked that some extracted algorithms satisfy the correctness condition in the frame of the *Coq* system [3].

1.1 Related Work

For an overview of the most common approaches used to tackle the synthesis problem, the reader may consult [13]. Synthesis methods and techniques similar to our proof-based approach are extensively presented in [12]. It can be noticed that most of the proof methods are based on expressive and undecidable logics that integrate induction principles.

The proof environments underlying deductive synthesis frameworks are usually supporting both automated and interactive proof methods. Those based on abstract datatype and computation refinements [2, 23] integrate techniques that are mainly executed manually and implemented by higher-order proof assistants like Isabelle/HOL [19] or more synthesis-oriented tools as Specware [20]. 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 [15]. 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 [8] to ensure the soundness of the validation step by certification with the Coq kernel. [7] 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. Generally speaking, generating proofs and implementing inference rules and strategies directly in Coq is a rather difficult task and does not fit for rapid prototyping and testing new ideas.

[5] `https://www.risc.jku.at/research/theorema/software/`

2 The Proof-based Synthesis Method

This section introduces the algorithm synthesis problem and presents the proof-based synthesis techniques which we use.

2.1 Our Approach

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

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

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.

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 trees X and Y into a new tree (if X, Y are sorted then the result is also sorted).

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

The formal definitions of these functions and predicates are:

Definition 1. $\underset{n,m,L,R,S}{\forall} \left(\begin{array}{c} RgM[\langle L, n, \varepsilon\rangle] = n \\ RgM[\langle L, n, \langle R, m, S\rangle\rangle] = RgM[\langle R, m, S\rangle] \end{array} \right)$

Definition 2. $\underset{n,m,L,R,S}{\forall}\left(\begin{array}{c} LfM[\langle\varepsilon,n,R\rangle]=n \\ LfM[\langle\langle L,n,R\rangle,m,S\rangle]=LfM[\langle L,n,R\rangle]\end{array}\right)$

Definition 3. $\underset{n,L,R,S}{\forall}\left(\begin{array}{c} Concat[\varepsilon,R]=R \\ Concat[\langle L,n,R\rangle,S]=\langle L,n,Concat[R,S]\rangle\end{array}\right)$

Definition 4. $\underset{L,m,R}{\forall}$

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

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

The functions *LfM* and *RgM* do not have a definition for the empty tree, however we assume that: $\underset{m}{\forall}\Big(RgM[\varepsilon]\leq m\leq LfM[\varepsilon]\Big)$.

A first simple property which can be proven inductively from Definition 3 is the following:

Property 1. $\underset{L}{\forall}\Big(Concat[L,\varepsilon]=L\Big)$

Moreover the following two properties are explicitly used in the proofs:

Property 2. $\underset{z,T}{\forall}\Big(IsSorted[T]\Longrightarrow(T\preceq z\Longleftrightarrow RgM[T]\leq z)\Big)$

Property 3. $\underset{z,T}{\forall}\Big(IsSorted[T]\Longrightarrow(z\preceq T\Longleftrightarrow z\leq LfM[T])\Big)$

All the statements used at object level in our experiments are formally just predicate logic formulae, however for this presentation we will call them differently depending on their role: a *definition* or an *axiom* is given as an initial piece of the theory, considered to hold; a *property* is a logical consequence of the definitions and axioms; a *proposition* is a formula which we sometimes assume, and sometimes prove, depending of the current experiment scenario; and a *conjecture* is something we want to prove.

The synthesis problem. 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: $\underset{X}{\forall}\underset{T}{\exists}(I[X]\Rightarrow O[X,T])$. 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 using the notations presented above (e.g., X is a tree). 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. 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 $Insert[a, A]$ (insert element a into sorted tree A, such that the result is sorted) and $Merge[A, B]$ (merge two sorted trees into a sorted tree). Their necessary properties are:

Proposition 1. $\underset{T}{\forall}\Big(Insert[n, T] \approx \langle \varepsilon, n, T\rangle\Big)$

Proposition 2. $\underset{T}{\forall}\Big(IsSorted[T] \Longrightarrow IsSorted[Insert[n, T]]\Big)$

Proposition 3. $\underset{L,R}{\forall}\Big((IsSorted[L] \wedge IsSorted[R]) \Longrightarrow IsSorted[Merge[L, R]]\Big)$

Furthermore the following properties are used to simplify the expression of some algorithms:

Proposition 4. $\underset{T}{\forall}\Big(Merge[T, \varepsilon] \approx T\Big)$

Proposition 5. $\underset{T}{\forall}\Big(Merge[\varepsilon, T] \approx T\Big)$

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.

2.2 Induction Principles and Algorithm Extraction

The illustration of the induction principles and algorithm extraction in this subsection is similar to the one from [9], 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 [21] 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 [1] 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 $P[\overline{t}]$, as *induction hypotheses*, any instance $P[\overline{t'}]$ of

$\underset{\overline{x}}{\forall} P[\overline{x}]$ as long as $\overline{t'} \ll_t \overline{t}$. 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 *number 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 in the tree (*Concat, Insert, Merge*).

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

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

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

In order to synthesize the sorting algorithm as a function $F[X]$, we consider the output condition $O[X, T] : \big(X \approx T \wedge \mathit{IsSorted}[T]\big)$. **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 $\Im_1$ such that $O[\varepsilon, \Im_1]$, then we know that $F[\varepsilon] = \Im_1$.

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 $T^* = \Im_2[n, L_0, R_0, T_1, T_2]$ (term depending on n, L_0, R_0, T_1 and T_2), then we know that $F[\langle L, n, R\rangle] = \Im_2[n, L, R, F[L], F[R]]$. ($T_1$ and T_2 are replaced by $F[L]$ and $F[R]$, respectively.)

The extracted algorithm from the proof is expressed as:

$$\underset{n,L,R}{\forall} \left(\begin{array}{c} F[\varepsilon] = \Im_1 \\ F[\langle L, n, R\rangle] = \Im_2[n, L, R, F[L], F[R]] \end{array} \right)$$

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 [6].

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:

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

The extracted algorithm is:

$$\underset{n,L,R}{\forall}\begin{pmatrix} F[\varepsilon] = \Im_1 \\ F[\langle L,n,\varepsilon] = \Im_3[n,L,F[L]] \\ F[\langle L,n,R\rangle] = \Im_5[n,L,R,F[\langle L,n,\varepsilon\rangle],F[R]] \end{pmatrix}$$

Induction-3:

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

In the formula above, L and R are assumed to be nonempty. 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:

$$\underset{n,a,b,A,B,C,D}{\forall}\begin{pmatrix} F[\varepsilon] = \Im_1 \\ F[\langle \varepsilon,n,\varepsilon\rangle] = \Im_2[n] \\ F[\langle\langle A,a,B\rangle,n,\varepsilon\rangle] = \Im_3[n,A,a,B,F[\langle A,a,B\rangle]] \\ F[\langle \varepsilon,n,\langle C,b,D\rangle\rangle] = \Im_4[n,C,b,D,F[\langle C,b,D\rangle]] \\ F[\langle\langle A,a,B\rangle,n,\langle C,b,D\rangle\rangle] = \Im_5[n,a,b,A,B,C,D, \\ F[\langle A,a,B\rangle],F[\langle C,b,D\rangle]] \end{pmatrix}$$

Induction schema discovery. In some examples (e.g. synthesis of $Merge[X,Y]$), it is not possible to find a witness term using only the constants and functions present in the proof situation. In such cases the prover allows the use of terms containing the function to be synthesized, by assuming that it fulfils the desired specification. However, the call of this function must apply to arguments which are strictly smaller (w.r.t. $\ll_t$) then the arguments of the main call of the function which is currently synthesized.

2.3 Special Inference Rules and Proof Strategies

SLD–resolution style. The *SLD–resolution style* of proving, introduced in [17] and described in [18], is used in Prolog. A first version of our provers uses a similar strategy, even as we do not classify formulae and we aim to generate natural style proofs. The use of Prolog–like reasoning is possible because we use first order predicate logic, and most of the formulae are essentially Horn: a possible empty conjunction of atoms implying one single atom. Moreover the goal is always a conjunction of atoms.

The inference rules for quantified formulae are: Skolemization for existential assumptions and universal goals, meta–variable for existential goal, and instantiation with backtracking for universal assumptions. By Skolemization new constants are introduced. 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. A universal assumptions is used 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. More details on these proving strategies are given in [12].

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 not able to generate complex proofs because the exhaustion of time or space resources.

In order to make the proving 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 for the theory of binary trees.

The inference rules which we present below are specific to the theory of binary trees because they are based on the definitions and on the properties of specific predicates and functions: $\approx$, *IsSorted*, *Concat*.

Some of the inference rules and strategies described in this subsection are similar to the ones for lists, see [9] and [10], but the ones described in this paper apply to binary trees and are more complex.

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

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

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 **IR-5** below).

IR-2: *Eliminate-Ground-Formulae-from-Goal.* If the goal contains a ground formula which is identical to (or an instance of) one of the assumptions, then this ground formula is deleted from the goal. Example: one of the assumptions is $\mathit{IsSorted}[T_2]$ and the goal is $\langle T_1,\ n, T_2\rangle \approx T^* \wedge \mathit{IsSorted}[T_1] \wedge \mathit{IsSorted}[T_2]$. The goal is transformed into: $\langle T_1, n, T_2\rangle \approx T^* \wedge \mathit{IsSorted}[T_1]$. This is a simple refinement of the Prolog style mechanism, which increases the efficiency of proving.

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: for instance in the example above T_1, T_2 correspond in the final algorithm to the recursive calls, so it is natural to try to include them in the goal, because we expect these recursive calls to be part of the synthesized algorithm.

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.

IR-3b: *Replace-Equivalent-Expression-in-Goal* This rule generalizes **IR-3a**, by allowing similar replacements when the expressions contain function symbols different from the tree constructor.

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.

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.* When the goal is of the form *Expression* $\approx T^* \wedge$ *IsSorted*$[T^*]$, the prover generates permutations of the list of nonempty 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*.

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 3) according to various criteria, and the remaining set of clauses can be used for the generation of various algorithms, each algorithm being composed of a certain subset of clauses.

Of course if the original expression contains more symbols then the resulting expressions will be more complicated and also quite many.

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.**

Remark: 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*[6] 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

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

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 (55).

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 Section 3.

IR-5: *Simple-Goal-Conditional-Assumption.* 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*, then 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 (see also strategy **S-3**). The reason for the selection of such formulae 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).

Example 1: The goal is: $m \leq n$.

Example 2: The goal is: $RgM[\langle A_2, z_2, B_2\rangle] \leq n \wedge m \leq LfM[\langle A_2, z_2, B_2\rangle]$.

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 more effective on goals. The application of this strategy for **Induction-1** is presented in Subsection 2.2. Note that for the soundness of the prover it is necessary to keep track of the order in which Skolem constants and meta–variables have been introduced, because a Skolem constant which cannot be generated before a certain meta–variable cannot be used in a solution for that meta–variable.

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

S-3: *Case-Distinction.* The prover generates several proof branches using rule **IR-4**, follows each branch in turn 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 $n \leq m$.

3 Combinatorial Technique

This section details the combinatorial technique which we use in order to synthesize the function for merging of sorted binary trees into a sorted one. Remarkably, merging requires a nested recursion, for which an appropriate induction principle is difficult

to guess. Our method is able to find it automatically by using a general Noetherian induction and this combinatorial technique.

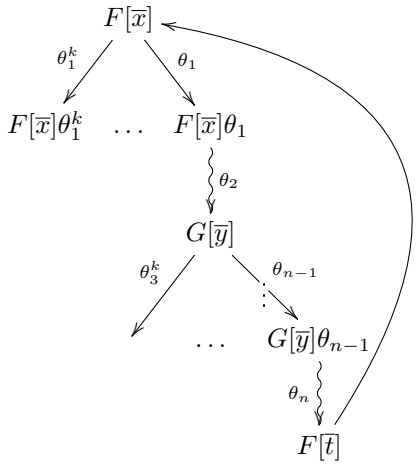
3.1 Refining Induction by Combinatorial Techniques and Lazy Reasoning

As shown e. g. in [21], 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.

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

In general, when we want to prove a formula $\underset{\overline{x}}{\forall} F[\overline{x}]$ by lazy induction, where $\overline{x}$ is a vector of variables, we start to instantiate variables from $\overline{x}$, then transform the resulted instances by using deductive rules. The instantiation and deduction steps can be intertwined up to the moment when instances of $F[\overline{x}]$ are encountered. An instance $F[\overline{t}]$ can be used as induction hypothesis for the induction case $F[\overline{x}]\theta$ if $\overline{t}$ is smaller than $\overline{x}\theta$.

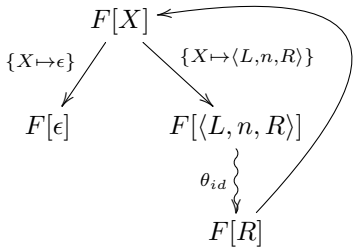
The substitution θ, called *cumulative* substitution, is built from the proof. To illustrate its computation, we represent the proof derivation as a tree for which the root node is labeled by $F[\overline{x}]$. Two kinds of non-root nodes are distinguished: instantiation nodes and deductive nodes. The instantiation nodes are direct successors of a node N labeled by a formula with free variables for which some of them are instantiated with terms whose variables are fresh. The set of instance formulas labeling all the instantiation nodes should *cover* the formula labeling N and can be built from the sort of the instantiated variables. For example, if N is labeled by the formula $F[X]$, a covering set of instance formulas is $\{F\{X \mapsto \epsilon\}, F\{X \mapsto \langle L, n, R\rangle\}\}$, where L, n, R are fresh variables. In the graphical representation of a proof tree, the relation between a node and its direct instantiation nodes are represented by downward solid arrows annotated by the corresponding instantiation substitution. The deductive nodes are direct successors of nodes to which a deductive operation has been applied. These relations are graphically represented as curly arrows annotated by identity substitutions. The cumulative substitution is the composition of the substitutions annotating the nodes from the path leading from the root node, in our case the node labeled by $F[\overline{x}]$, to the node labeled by the induction hypothesis, in our case $F[\overline{t}]$. This scenario can be illustrated as below:



In our scenario, $F[\overline{t}]$ is an instance of $F[\overline{x}]$. 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 1. By lazy induction, one can benefit of more effective induction reasoning, involving only useful induction hypotheses.

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

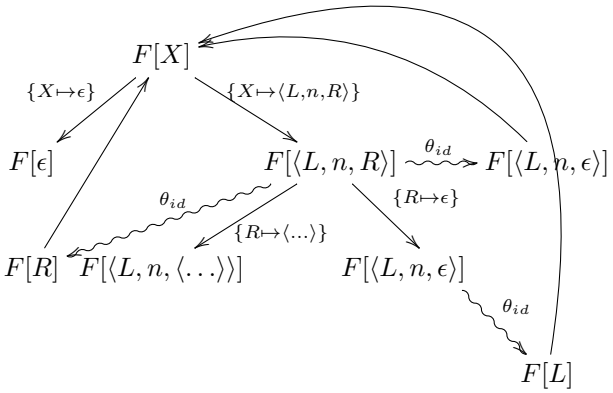


where θ_{id} is the identity substitution $\{L \mapsto L; n \mapsto n; R \mapsto R\}$. $F[R]$ can be used as induction hypothesis in the proof of the case $F[\langle L, n, R\rangle]$ because R has a number of elements smaller than $\langle L, n, R\rangle$.

The corresponding explicit induction principle is:

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

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



The corresponding explicit induction principle is:

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

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

3.2 Synthesis of *Merge*

The prover automatically generates the proof of Conjecture 4, 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 $\{T^* \to \langle Concat[\varepsilon, R_0]\}$, which is $\{T^* \to R_0\}$.
Induction step:

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

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

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

and the induction goal (to prove) is:

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

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 microatoms:

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

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

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

The goal is:

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

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

Since **IR-3b** can be applied on (51) in two different ways, we generate two alternatives:

Alternative-1: By applying **IR-3b** using (50), the goal is transformed into:

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

In this moment the prover uses the *combinatorial technique*, namely it applies **IR-4** and generates all the permutations of $\langle T_1, n, R\rangle$ and to each permutation all possible expressions containing all the symbols in the list, composed by using the tree constructors and the functions $Concat, Insert$, 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 because the function *Merge* is symmetric and the conditions are not influenced by the order of its arguments.

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

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

Each expression is further processed by replacing each occurrence of T_1 by $Merge[L, S]$ — according to (1), 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).

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

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

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

Algorithm 1

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

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

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

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

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

$$\{\} \Longrightarrow Merge[Insert[n, L], Merge[R, S]] \tag{26}$$
$$\{\} \Longrightarrow Merge[\langle L, n, \varepsilon\rangle, Merge[R, S]] \tag{27}$$
$$\{\} \Longrightarrow Insert[n, Merge[L, Merge[R, S]]] \tag{28}$$
$$\{\} \Longrightarrow Merge[Insert[n, Merge[R, S]], L] \tag{29}$$
$$\{L \ll S\} \Longrightarrow \langle Insert[n, Concat[L, Merge[R, S]]] \tag{30}$$
$$\{L \ll S\} \Longrightarrow Concat[L, Insert[n, Merge[R, S]]] \tag{31}$$
$$\{n \preceq S\} \Longrightarrow Merge[\langle \varepsilon, n, Merge[R, S]\rangle, L] \tag{32}$$
$$\{n \preceq S\} \Longrightarrow \langle L, n, Merge[R, S]\rangle \tag{33}$$

The most important clause generated here and which is the analogous of (19), is (28).

Any algorithm composed from such conditional clauses must fulfill two important conditions: *proper ordering of clauses* and *coverage of all cases*. One possible algorithm is based on clauses (33),(31),(28) (in this order):

Algorithm 2

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

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

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

In the resulting algorithms the conditions are tested using the functions *LfM, RgM*. In a program with several clauses, multiple calls to these functions can be easily avoided by computing their values before the evaluation of the clauses (the

functional `let` from `lisp`). However their use still remains quite expensive, because the recursive calls will repeat the descending of the tree. This problem (suggested by the automatically generated algorithms) can be solved by changing the data structure: one can store the respective values in each node of the tree (preprocessing for computing them will be linear), and then *LfM, RgM* will be evaluated in constant time.

4 Experiments

4.1 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:

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

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: $\varepsilon \approx T^* \wedge \mathit{IsSorted}[T^*]$.
One obtains the substitution $\{T^* \to \varepsilon\}$ and the new goal is $\mathit{IsSorted}[\varepsilon]$, which is true by Definition 4.

Base case 2: Prove: $\langle \varepsilon, n, \varepsilon \rangle \approx T^* \wedge \mathit{IsSorted}[T^*]$.
One obtains the substitution $\{T^* \to \langle \varepsilon, n, \varepsilon \rangle\}$. The new goal is $\mathit{IsSorted}[\langle \varepsilon, n, \varepsilon \rangle]$ which is true by Definition 4.

Induction case 1: Assume:

$$\underset{T}{\exists}(L_0 \approx T \wedge \mathit{IsSorted}[T]) \tag{34}$$

and prove:

$$\underset{T}{\exists}(\langle L_0, n, \varepsilon \rangle \approx T \wedge \mathit{IsSorted}[T]) \tag{35}$$

Apply **S-1** on (34) and (35) to eliminate the existential quantifiers. The induction hypothesis are:

$$L_0 \approx T_1, \quad \mathit{IsSorted}[T_1] \tag{36}$$

and the goal is:

$$\langle L_0, n, \varepsilon \rangle \approx T^* \wedge \mathit{IsSorted}[T^*] \tag{37}$$

Apply **IR-3** and rewrite our goal (37) by using the first conjunct of the assumption (36). The goal becomes:

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

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

$$\mathit{IsSorted}[\langle T_1, n, \varepsilon \rangle] \tag{39}$$

Apply **IR-1** on (39) and prove:

$$IsSorted[T_1] \wedge RgM[T_1] \leq n \tag{40}$$

Apply **IR-2** using (36) and the new goal is:

$$RgM[T_1] \leq n \tag{41}$$

Apply **IR-5** and the goal (41) 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:

$$n \leq LfM[T_1] \tag{42}$$

which becomes the conditional assumption on this branch.
Alternative-3: Since the disjunction of the conditions 41 and 42 is not provable, the prover generates a further alternative. This depends on the synthesis scenario (see the end of Section 2.1). 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 3 - displayed in Subsection 4.3 - is generated, and this is used for the synthesis of *Insert* as shown in Subsection 4.3. 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:

$$L_1 \approx T_3, \quad IsSorted[T_3], \quad R_1 \approx T_4, \quad IsSorted[T_4] \tag{43}$$

and prove:

$$\langle L_1, n, R_1 \rangle \approx T^* \wedge IsSorted[T^*] \tag{44}$$

Apply **IR-3** and rewrite our goal (44) by using the first and the third conjunct of the assumption (43) and the new goal is:

$$\langle T_3, n, T_4 \rangle \approx T^* \wedge IsSorted[T^*] \tag{45}$$

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 4 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):

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

4.2 Additional Certification of the Synthesized Algorithm F_1

Even if our approach theoretically guarantees the soundness of the synthesized algorithms, the implementation of the presented rules in *Theorema* is error-prone. To check the soundness of the implementation, we have mechanically verified that the algorithm F_1 satisfies the correctness condition, by using the Coq proof assistant (`https://coq.inria.fr`). The Coq formalization of the *LfM* and *RfM* functions has slightly changed from the partial definitions given here, as Coq requires that the functions be total. The conversion into total functions is possible if the components of the triplet given as argument are represented as the new arguments, as below.

Definition 5. $\underset{n,m,L,R,S}{\forall}\left(\begin{array}{c} RgM[L, n, \varepsilon] = n \\ RgM[L, n, \langle R, m, S\rangle] = RgM[R, m, S] \end{array}\right)$

Definition 6. $\underset{n,m,L,R,S}{\forall}\left(\begin{array}{c} LfM[\varepsilon, n, R] = n \\ LfM[\langle L, n, R\rangle, m, S] = LfM[L, n, R] \end{array}\right)$

The proof effort was non-trivial, involving significant user interaction. The certification proofs used rules and proof strategies completely different from those generating the synthesized algorithms, requiring additionally 2 induction schemas and 15 lemmas. The full Coq script can be found at: `http://web.info.uvt.ro/~idramnesc/ICTAC2015/coq.v`

4.3 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 LHS of the implication consists of the current assumptions and the RHS 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 [12] and is an extension of the method presented in [4]. 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 4.1, if the functions *Insert* and *Merge* are not present in the knowledge base, then the prover generates automatically from (43) and (45) the following conjecture:

$$\textit{Conjecture 2.} \quad \underset{\substack{n,L,R \\ IsSorted[L],IsSorted[R]}}{\forall} \; \underset{T}{\exists}\Big(\langle L,n,R\rangle \approx T \wedge IsSorted[T]\Big)$$

We manually decompose this conjecture in two sub-problems:

$$\textit{Conjecture 3.} \quad \underset{\substack{n,R \\ IsSorted[R]}}{\forall} \; \underset{T}{\exists}\Big(\langle \varepsilon,n,R\rangle \approx T \wedge IsSorted[T]\Big)$$

$$\textit{Conjecture 4.} \quad \underset{\substack{L,R \\ IsSorted[L],IsSorted[R]}}{\forall} \; \underset{T}{\exists}\Big(Concat[L,R] \approx T \wedge IsSorted[T]\Big)$$

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

The prover automatically generates the proof of Conjecture 3 by applying **Induction-1** (on the second argument) and the specific inference rules and strategies from Subsection 2.3. 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 witness found is $\{T^* \rightarrow \langle \varepsilon,n,\varepsilon\rangle\}$.
Induction step: We assume:

$$IsSorted[L] \Longrightarrow (\langle \varepsilon,n,L\rangle \approx T_1 \wedge IsSorted[T_1]) \tag{46}$$

$$IsSorted[R] \Longrightarrow (\langle \varepsilon,n,R\rangle \approx T_2 \wedge IsSorted[T_2]) \tag{47}$$

and we prove:

$$IsSorted[\langle L,m,R\rangle] \Longrightarrow (\langle \varepsilon,n,\langle L,m,R\rangle\rangle \approx T^* \wedge IsSorted[T^*]) \tag{48}$$

We prove the RHS of the above implication, by assuming the LHS, which using **IR-1** is decomposed into:

$$IsSorted[L], \quad IsSorted[R], \quad RgM[L] \leq m, \quad m \leq LfM[R], \quad L \preceq m, \quad m \preceq R \tag{49}$$

Using *modus ponens* from (49) by (46) and (47) we obtain:

$$\langle \varepsilon, n, L\rangle \approx T_1,\ IsSorted[T_1],\ \langle \varepsilon, n, R\rangle \approx T_2,\ IsSorted[T_2] \tag{50}$$

The goal is:

$$\langle \varepsilon, n, \langle L, m, R\rangle\rangle \approx T^* \wedge IsSorted[T^*] \tag{51}$$

Since **IR-3a** can be applied on (51) in two different ways, we generate two alternatives:
Alternative-1: By applying **IR-3a** using the first two assumptions from (50), the goal is transformed into:

$$\langle T_1, m, R\rangle \approx T^* \wedge IsSorted[T^*] \tag{52}$$

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:

$$IsSorted[T_1] \wedge IsSorted[R] \wedge RgM[T_1] \leq m \wedge m \leq LfM[R] \tag{53}$$

Eliminate the first two conjuncts of the goal (apply **IR-2** using (50), (49)) and the new goal is: $RgM[T_1] \leq m \wedge m \leq LfM[R]$. Apply **IR-3c** using (50) and the goal becomes: $n \leq m \wedge L \preceq m \wedge m \leq LfM[R]$. Apply **IR-2** using (49) 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 (50) the new goal is:

$$\langle L, m, T_2\rangle \approx T^* \wedge IsSorted[T^*] \tag{54}$$

Similar to the previous case and by using Property 2 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:

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

In a similar manner the synthesis of *Merge* is also performed, by proving Conjecture 4 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 need to allow for the function *Merge* to be used in the expressions generated by various permutations, as well as its property, although we want to synthesize exactly this function. The algorithms generated are still terminating, because we allow the use of *Merge* only when the first argument is smaller (has fewer elements) than the argument of the main call of the function. The proof uses the inference **IR-4** and the strategy **S-3**, which are more complex and are presented in detail in Section 3, together with the respective proof. Similarly to the situation in the synthesis of *Sort-1* (see proof step after formula (45)) 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:

$$\underset{n,L,R,S}{\forall}\left(\begin{array}{r} Merge[\varepsilon,R]=R \\ Merge[\langle L,n,R\rangle,S]=Insert[n,Merge[L,Merge[R,S]]] \end{array}\right) \tag{55}$$

4.4 Synthesis of Other Sorting Algorithms

Sort-2. The prover generated automatically the proof of Conjecture 1 by applying **Induction-2** and by using the current knowledge base (Definition 4, Proposition 2, and 3), including the following property:

Proposition 6. $\underset{n,L,R,A,B}{\forall}((\langle L,n,\varepsilon\rangle \approx A \wedge R \approx B) \Longrightarrow \langle L,n,R\rangle \approx Merge[A,B])$

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

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

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:

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

Sort-4. The prover 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 applies the inference rule **IR-4** which generates permutations.

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

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

Sort-5. The prover 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 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 *Theorema* are fully presented in the next section.

The following table presents the synthesized sorting algorithms. For each of them Conjecture 1 has been proved using the induction principles from the first column. The second column specifies the auxiliary function used and the third column shows whether the rule **IR-4** (which generates the permutations and witnesses) is used or not.

Induction principle	Auxiliary used functions	Uses **IR-4**	Extracted algorithm
Induction-2	`LfM, RgM, Insert, Merge`	No	F_2
Induction-3	`LfM, RgM, Insert, Merge`	No	F_1
	`LfM, RgM, Insert, Merge`	Yes	F_4
	`LfM, RgM, Insert, Concat`	No	F_3
	`LfM, RgM, Insert, Concat`	Yes	F_5

5 Theorema files

This section includes the files from the *Theorema* system. The file demo-trees.nb is the user's file. In this file: one loads the *Theorema* system; one loads the prover (TreeSynthesizer) which the authors implemented in the *Theorema*, and which is used for proving; one adds the definitions and the properties; one calls the loaded prover to generate automatically some proofs; after the proofs succeed one calls the extractor which extracts from the proofs the corresponding algorithms, and one can compute with the extracted algorithms. All the other files (Sort-1-proof.nb, Insert-proof.nb, Sort-4-proof.nb, Sort-2-proof.nb, Sort-3-proof.nb, and Sort-5.nb) are generated automatically by our prover (TreeSynthesizer).

```
Needs["Theorema`"]

Get["Theorema`Provers`UserProvers`TreeSynthesizer`"]
```

TS_In[1406]:=
```
Use[⟨Built-in["Connectives"], Built-in["Numbers"]⟩]
```

■ Knowledge base

TS_In[312]:=
```
SetGlobals[TraceLevel⟶0, FormatMetas→"Subscripted"];
```

TS_In[313]:=
```
Definition["Right Most Element", any[n, m, L, S, R],
                RgM[⟨L, n, ϵ⟩] = n
 RgM[⟨L, n, ⟨R, m, S⟩⟩] = RgM[⟨R, m, S⟩]]
```

TS_In[314]:=
```
Definition["Left Most Element", any[n, m, L, S, R],
                LfM[⟨ϵ, n, R⟩] = n
 LfM[⟨⟨L, n, R⟩, m, S⟩] = LfM[⟨L, n, R⟩]]
```

TS_In[315]:=
```
Definition["Concat", any[n, L, R, S],
                 Concat[ϵ, R] = R
 Concat[⟨L, n, R⟩, S] = ⟨L, n, Concat[R, S]⟩]
```

TS_In[1433]:=
```
Definition["Merge", any[n, L, R, S],
                       Merge[ϵ, R] = R
 Merge[⟨L, n, R⟩, S] = Insertion[n, Merge[Merge[L, R], S]]]
```

TS_In[317]:=
```
Definition["IsSorted", any[m, L, R],
                                  IsSorted[ϵ]
 (IsSorted[L] ⋀ IsSorted[R] ⋀ RgM[L] ≤ m ⋀ m ≤ LfM[R]) ⟹ IsSorted[⟨L, m, R⟩]
 ]
```

TS_In[318]:=
```
Proposition["RgM empty",
 ⋀m (RgM[ϵ] ≤ m)]
```

TS_In[319]:=
```
Proposition["LfM empty",
 ⋀m (m ≤ LfM[ϵ])]
```

TS_In[320]:=

```
Proposition["1",
  ∀n,L (⟨L, n, ϵ⟩ ≈ Insertion[n, L])]
```

TS_In[322]:=

```
Proposition["1-1",
  ∀n,L,R (⟨L, n, R⟩ ≈ Insertion[n, Merge[L, R]])]
```

TS_In[323]:=

```
Proposition["1-2",
  ∀n,L,R (⟨L, n, R⟩ ≈ Insertion[n, Concat[L, R]])]
```

TS_In[325]:=

```
Proposition["1-5",
  ∀n,L,R,A,B ((⟨L, n, ϵ⟩ ≈ A ⋀ R ≈ B) ⟹ (⟨L, n, R⟩ ≈ Merge[A, B]))]
```

TS_In[326]:=

```
Proposition["1-6",
  ∀n,L,R,A,B,C ((L ≈ A ⋀ R ≈ B ⋀ Concat[L, R] ≈ C) ⟹
    (⟨A, n, B⟩ ≈ Insertion[n, C]))]
```

TS_In[327]:=

```
Proposition["2",
  ∀n,L (IsSorted[L] ⟹ IsSorted[Insertion[n, L]])]
```

TS_In[328]:=

```
Proposition["3",
  ∀L,R ((IsSorted[L] ⋀ IsSorted[R]) ⟹ IsSorted[Merge[L, R]])]

Proposition["3-1",
  ∀L,R ((IsSorted[L] ⋀ IsSorted[R] ⋀ L ⋘ R) ⟹ IsSorted[Concat[L, R]])]
```

TS_In[330]:=

```
Proposition["4",
  ∀T (Merge[T, ϵ] ≈ T)]
```

TS_In[331]:=

```
Proposition["5",
  ∀T (Merge[ϵ, T] ≈ T)]
```

TS_In[332]:=

```
Proposition["Problem of Sorting",
 ∀X ∃T (X ≈ T ⋀ IsSorted[T])]
```

■ Synthesis of Sort-1

TS_In[1436]:=

```
Prove[Proposition["Problem of Sorting"],
   by → TreeSynthesizer, SearchDepth → 25,
   using → ⟨Definition["IsSorted"], Proposition["RgM empty"],
     Proposition["LfM empty"], Proposition["1-1"], Proposition["2"],
     Proposition["3"]⟩, TransformBy → ProofSimplifier,
   TransformerOptions → {branches -> Proved},
   ProverOptions → {Induction3 → True}] // Last // Timing
```

TS_Out[1436]=

```
{4.634, proved}
```

The proof is generated in a separate window (which is the Theorema proof object). Please see Sort-1-proof.nb

From this proof we automatically extract the corresponding algorithm by calling the function AlgorithmFromProof

```
AlgorithmFromProof["Induction3", $TmaProofObject]
```

TS_In[1437]:=

```
Algorithm["Sort-1", any[n, L, R],

                    F1[ϵ] = ϵ
              F1[⟨ϵ, n, ϵ⟩] = ⟨ϵ, n, ϵ⟩
  (RgM[F1[L]] ≤ n) ⟹ (F1[⟨L, n, ϵ⟩] = ⟨F1[L], n, ϵ⟩)
  (n ≤ LfM[F1[L]]) ⟹ (F1[⟨L, n, ϵ⟩] = ⟨ϵ, n, F1[L]⟩)
        F1[⟨L, n, ϵ⟩] = Insertion[n, F1[L]]
  (n ≤ LfM[F1[R]]) ⟹ (F1[⟨ϵ, n, R⟩] = ⟨ϵ, n, F1[R]⟩)
  (RgM[F1[R]] ≤ n) ⟹ (F1[⟨ϵ, n, R⟩] = ⟨F1[R], n, ϵ⟩)
        F1[⟨ϵ, n, R⟩] = Insertion[n, F1[R]]
 F1[⟨L, n, R⟩] = Insertion[n, Merge[F1[L], F1[R]]]]
```

▪ Compute with Sort-1

TS_In[1438]:=
```
Compute[F1[⟨⟨ϵ, 9, ϵ⟩, 18, ⟨⟨ϵ, 19, ϵ⟩, 14, ϵ⟩⟩],
 using → ⟨Algorithm["Sort-1"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Merge"]⟩]
```

TS_Out[1438]=
```
⟨⟨ϵ, 9, ϵ⟩, 14, ⟨⟨ϵ, 18, ϵ⟩, 19, ϵ⟩⟩
```

TS_In[1439]:=
```
Compute[F1[⟨⟨⟨⟨ϵ, 100, ϵ⟩, 10, ϵ⟩, 10, ⟨ϵ, 11, ϵ⟩⟩, 3, ϵ⟩],
 using → ⟨Algorithm["Sort-1"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Merge"]⟩]
```

TS_Out[1439]=
```
⟨ϵ, 3, ⟨⟨⟨ϵ, 10, ϵ⟩, 10, ϵ⟩, 11, ⟨ϵ, 100, ϵ⟩⟩⟩
```

▪ Synthesis of Insertion

TS_In[1444]:=
```
Proposition["Conjecture-1",
  ∀_{n,R, IsSorted[R]} ∃_T (⟨ϵ, n, R⟩ ≈ T ⋀ IsSorted[T])]
```

TS_In[1445]:=
```
Proposition["Conjecture 1",
  ∀_{n,R} (IsSorted[R] ⟹ ∃_T (⟨ϵ, n, R⟩ ≈ T ⋀ IsSorted[T]))]
```

TS_In[2064]:=
```
Prove[Proposition["Conjecture 1"],
   by → TreeSynthesizer, SearchDepth → 35, using →
    ⟨Definition["IsSorted"], Proposition["1"], Proposition["1-1"],
     Proposition["2"], Proposition["3"], Proposition["RgM empty"],
     Proposition["LfM empty"]⟩, TransformBy → ProofSimplifier,
   TransformerOptions → {branches -> Proved}, ProverOptions →
    {Induction1 → True, ExtendAssm → True}] // Last // Timing
```

TS_Out[2064]=
```
{2.247, proved}
```

The proof is generated in a separate window. Please see Insert-proof.nb

```
AlgorithmFromProof["Induction1", $TmaProofObject]
```

The extracted algorithm from the proof is :

```
TS_In[1447]:=
Definition["Insert", any[n, m, L, R],
                    Insertion[n, ϵ] = ⟨ϵ, n, ϵ⟩
 (n ≤ m) ⟹ (Insertion[n, ⟨L, m, R⟩] = ⟨Insertion[n, L], m, R⟩)
 (m ≤ n) ⟹ (Insertion[n, ⟨L, m, R⟩] = ⟨L, m, Insertion[n, R]⟩)]
```

▪ Compute with Insert

```
TS_In[1448]:=
Compute[Insertion[2, ⟨ϵ, 10, ϵ⟩], using → Definition["Insert"]]

TS_Out[1448]=
⟨⟨ϵ, 2, ϵ⟩, 10, ϵ⟩

TS_In[1449]:=
Compute[Insertion[12, ⟨ϵ, 10, ϵ⟩], using → Definition["Insert"]]

TS_Out[1449]=
⟨ϵ, 10, ⟨ϵ, 12, ϵ⟩⟩

TS_In[1450]:=
Compute[Insertion[10, ⟨ϵ, 10, ϵ⟩], using → Definition["Insert"]]

TS_Out[1450]=
⟨⟨ϵ, 10, ϵ⟩, 10, ϵ⟩

TS_In[1451]:=
Compute[Insertion[8, ⟨⟨ϵ, 9, ⟨ϵ, 12, ϵ⟩⟩, 10, ϵ⟩],
 using → Definition["Insert"]]

TS_Out[1451]=
⟨⟨⟨ϵ, 8, ϵ⟩, 9, ⟨ϵ, 12, ϵ⟩⟩, 10, ϵ⟩

TS_In[1452]:=
Compute[Insertion[18, ⟨⟨ϵ, 9, ⟨ϵ, 12, ϵ⟩⟩, 10, ϵ⟩],
 using → Definition["Insert"]]

TS_Out[1452]=
⟨⟨ϵ, 9, ⟨ϵ, 12, ϵ⟩⟩, 10, ⟨ϵ, 18, ϵ⟩⟩
```

■ Synthesis of Sort-4

```
TS_In[1440]:=
Prove[Proposition["Problem of Sorting"],
   by → TreeSynthesizer, SearchDepth → 25,
   using → ⟨Definition["IsSorted"], Proposition["1-1"],
     Proposition["2"], Proposition["3"], Proposition["RgM empty"],
     Proposition["LfM empty"]⟩, TransformBy → ProofSimplifier,
   TransformerOptions → {branches -> Proved},
   ProverOptions → {Induction3 → True, IR4 → True}] // Last // Timing
```

```
TS_Out[1440]=
{5.179, proved}
```

The automated proof is generated in a separate window. Please see Sort-4-proof.nb.
The difference between Sort-1 and Sort-4 is at the last branch, where IR-4 generates permutations of a tree and Sort-4 has three branches.

From this proof we automatically extract the following algorithm:

```
AlgorithmFromProof2["Induction3", $TmaProofObject]
```

```
TS_In[1441]:=
Algorithm["Sort-4", any[n, L, R],
                                F4[ϵ] = ϵ
                          F4[⟨ϵ, n, ϵ⟩] = ⟨ϵ, n, ϵ⟩
              (RgM[F4[L]] ≤ n) ⟹ (F4[⟨L, n, ϵ⟩] = ⟨F4[L], n, ϵ⟩)
              (n ≤ LfM[F4[L]]) ⟹ (F4[⟨L, n, ϵ⟩] = ⟨ϵ, n, F4[L]⟩)
                    F4[⟨L, n, ϵ⟩] = Insertion[n, F4[L]]
              (n ≤ LfM[F4[R]]) ⟹ (F4[⟨ϵ, n, R⟩] = ⟨ϵ, n, F4[R]⟩)
              (RgM[F4[R]] ≤ n) ⟹ (F4[⟨ϵ, n, R⟩] = ⟨F4[R], n, ϵ⟩)
                    F4[⟨ϵ, n, R⟩] = Insertion[n, F4[R]]
     (RgM[F4[L]] ≤ n ⋀ n ≤ LfM[F4[R]]) ⟹ (F4[⟨L, n, R⟩] = ⟨F4[L], n, F4[R]⟩)
     (RgM[F4[R]] ≤ n ⋀ n ≤ LfM[F4[L]]) ⟹ (F4[⟨L, n, R⟩] = ⟨F4[R], n, F4[L]⟩)
           F4[⟨L, n, R⟩] = Insertion[n, Merge[F4[L], F4[R]]]
]
```

■ Compute with Sort-4

```
TS_In[1442]:=
Compute[F4[⟨⟨ϵ, 9, ϵ⟩, 18, ⟨⟨ϵ, 19, ϵ⟩, 14, ϵ⟩⟩],
 using → ⟨Algorithm["Sort-4"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Merge"]⟩]
```

```
TS_Out[1442]=
⟨⟨⟨ϵ, 9, ϵ⟩, 14, ⟨ϵ, 18, ϵ⟩⟩, 19, ϵ⟩
```

```
TS_In[1443]:=
Compute[F4[⟨⟨⟨⟨ϵ, 100, ϵ⟩, 10, ϵ⟩, 10, ⟨ϵ, 11, ϵ⟩⟩, 3, ϵ⟩],
 using → ⟨Algorithm["Sort-4"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Merge"]⟩]
```

```
TS_Out[1443]=
⟨⟨⟨ϵ, 3, ϵ⟩, 10, ϵ⟩, 10, ⟨⟨ϵ, 11, ϵ⟩, 100, ϵ⟩⟩
```

■ Synthesis of Sort-2

```
TS_In[1453]:=
Prove[Proposition["Problem of Sorting"],
   by → TreeSynthesizer, SearchDepth → 25,
   using → ⟨Definition["IsSorted"], Proposition["1-5"],
     Proposition["2"], Proposition["3"]⟩, TransformBy →
    ProofSimplifier, TransformerOptions → {branches -> Proved},
   ProverOptions → {Induction2 → True}] // Last // Timing
```

```
TS_Out[1453]=
{1.841, proved}
```

The proof is generated in a separate window. Please see Sort-2-proof.nb

```
AlgorithmFromProof["Induction2", $TmaProofObject]
```

The extracted algorithm from the proof is :

```
TS_In[1454]:=
Algorithm["Sort-2", any[n, L, R],

                    F2[ϵ] = ϵ
 (RgM[F2[L]] ≤ n) ⟹ (F2[⟨L, n, ϵ⟩] = ⟨F2[L], n, ϵ⟩)
 (n ≤ LfM[F2[L]]) ⟹ (F2[⟨L, n, ϵ⟩] = ⟨ϵ, n, F2[L]⟩)
       F2[⟨L, n, ϵ⟩] = Insertion[n, F2[L]]
   F2[⟨L, n, R⟩] = Merge[F2[⟨L, n, ϵ⟩], F2[R]]
```

■ Compute with Sort-2

```
TS_In[1455]:=
Compute[F2[⟨⟨ϵ, 9, ϵ⟩, 18, ⟨⟨ϵ, 19, ϵ⟩, 14, ϵ⟩⟩],
 using → ⟨Algorithm["Sort-2"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Merge"]⟩]
```

```
TS_Out[1455]=
⟨⟨ϵ, 9, ϵ⟩, 14, ⟨⟨ϵ, 18, ϵ⟩, 19, ϵ⟩⟩
```

TS_In[1456]:=

```
Compute[F2[⟨⟨⟨⟨ϵ, 100, ϵ⟩, 10, ϵ⟩, 10, ⟨ϵ, 11, ϵ⟩⟩, 3, ϵ⟩],
 using → ⟨Algorithm["Sort-2"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Merge"]⟩]
```

TS_Out[1456]=

```
⟨ϵ, 3, ⟨⟨⟨ϵ, 10, ϵ⟩, 10, ϵ⟩, 11, ⟨ϵ, 100, ϵ⟩⟩⟩
```

■ Synthesis of Sort-3

TS_In[1526]:=

```
Prove[Proposition["Problem of Sorting"],
   by → TreeSynthesizer, SearchDepth → 25,
   using → ⟨Definition["IsSorted"], Proposition["RgM empty"],
     Proposition["LfM empty"], Proposition["2"], Proposition["1-6"]⟩,
   TransformBy → ProofSimplifier, TransformerOptions →
    {branches -> Proved}, ProverOptions →
    {Induction3 → True, ExtendAssm → True}] // Last // Timing
```

TS_Out[1526]=

```
{5.007, proved}
```

Please see Sort-3-proof.nb

```
AlgorithmFromProof["Induction3", $TmaProofObject]
```

The extracted algorithm from the proof is :

TS_In[1527]:=

```
Algorithm["Sort-3", any[n, L, R],

                    F3[ϵ] = ϵ
               F3[⟨ϵ, n, ϵ⟩] = ⟨ϵ, n, ϵ⟩
 (RgM[F3[L]] ≤ n) ⟹ (F3[⟨L, n, ϵ⟩] = ⟨F3[L], n, ϵ⟩)
 (n ≤ LfM[F3[L]]) ⟹ (F3[⟨L, n, ϵ⟩] = ⟨ϵ, n, F3[L]⟩)
       F3[⟨L, n, ϵ⟩] = Insertion[n, F3[L]]
 (n ≤ LfM[F3[R]]) ⟹ (F3[⟨ϵ, n, R⟩] = ⟨ϵ, n, F3[R]⟩)
 (RgM[F3[R]] ≤ n) ⟹ (F3[⟨ϵ, n, R⟩] = ⟨F3[R], n, ϵ⟩)
       F3[⟨ϵ, n, R⟩] = Insertion[n, F3[R]]
  F3[⟨L, n, R⟩] = Insertion[n, F3[Concat[L, R]]]
```

▪ Compute with Sort-3

```
TS_In[1528]:=
Compute[F3[⟨⟨ϵ, 9, ϵ⟩, 18, ⟨⟨ϵ, 19, ϵ⟩, 14, ϵ⟩⟩],
 using → ⟨Algorithm["Sort-3"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Concat"]⟩]
```

```
TS_Out[1528]=
⟨⟨⟨ϵ, 9, ϵ⟩, 14, ⟨ϵ, 18, ϵ⟩⟩, 19, ϵ⟩
```

```
TS_In[1529]:=
Compute[F3[⟨⟨⟨⟨ϵ, 100, ϵ⟩, 10, ϵ⟩, 10, ⟨ϵ, 11, ϵ⟩⟩, 3, ϵ⟩],
 using → ⟨Algorithm["Sort-3"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Concat"]⟩]
```

```
TS_Out[1529]=
⟨⟨⟨⟨⟨ϵ, 3, ϵ⟩, 10, ϵ⟩, 10, ϵ⟩, 11, ϵ⟩, 100, ϵ⟩
```

▪ Synthesis of Sort-5

The proof for Sort-5 is similar to the proof of Sort-3, but for Sort-5 one uses IR-4 for generating the permutations of a tree.

```
Prove[Proposition["Problem of Sorting"],
   by → TreeSynthesizer, SearchDepth → 25,
   using → ⟨Definition["IsSorted"], Proposition["RgM empty"],
     Proposition["LfM empty"], Proposition["1-6"], Proposition["2"]⟩,
   TransformBy → ProofSimplifier,
   TransformerOptions → {branches -> Proved},
   ProverOptions → {Induction3 → True, IR4 → True}] // Last // Timing
```

```
TS_Out[1420]=
{5.179, proved}
```

Please see sort-5-proof.nb

```
AlgorithmFromProof["Induction3", $TmaProofObject]
```

The extracted algorithm from the proof is :

```
TS_In[1530]:=
Algorithm["Sort-5", any[n, L, R],
                                F5[ϵ] = ϵ
                            F5[⟨ϵ, n, ϵ⟩] = ⟨ϵ, n, ϵ⟩
                  (RgM[F5[L]] ≤ n) ⟹ (F5[⟨L, n, ϵ⟩] = ⟨F5[L], n, ϵ⟩)
                  (n ≤ LfM[F5[L]]) ⟹ (F5[⟨L, n, ϵ⟩] = ⟨ϵ, n, F5[L]⟩)
                        F5[⟨L, n, ϵ⟩] = Insertion[n, F5[L]]
                  (n ≤ LfM[F5[R]]) ⟹ (F5[⟨ϵ, n, R⟩] = ⟨ϵ, n, F5[R]⟩)
                  (RgM[F5[R]] ≤ n) ⟹ (F5[⟨ϵ, n, R⟩] = ⟨F5[R], n, ϵ⟩)
                        F5[⟨ϵ, n, R⟩] = Insertion[n, F5[R]]
       (RgM[F5[L]] ≤ n ⋀ n ≤ LfM[F5[R]]) ⟹ (F5[⟨L, n, R⟩] = ⟨F5[L], n, F5[R]⟩)
       (RgM[F5[R]] ≤ n ⋀ n ≤ LfM[F5[L]]) ⟹ (F5[⟨L, n, R⟩] = ⟨F5[R], n, F5[L]⟩)
                  F5[⟨L, n, R⟩] = Insertion[n, F5[Concat[L, R]]]
]
```

The algorithm is similar to F_3 except the last case where F_5 has three branches.

▪ Compute with Sort-5

```
TS_In[1531]:=
Compute[F5[⟨⟨ϵ, 9, ϵ⟩, 18, ⟨⟨ϵ, 19, ϵ⟩, 14, ϵ⟩⟩],
 using → ⟨Algorithm["Sort-5"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Concat"]⟩]
```

```
TS_Out[1531]=
⟨ϵ, 9, ⟨⟨ϵ, 14, ⟨ϵ, 18, ϵ⟩⟩, 19, ϵ⟩⟩
```

```
TS_In[1532]:=
Compute[F5[⟨⟨⟨⟨ϵ, 100, ϵ⟩, 10, ϵ⟩, 10, ⟨ϵ, 11, ϵ⟩⟩, 3, ϵ⟩],
 using → ⟨Algorithm["Sort-5"], Definition["Right Most Element"],
   Definition["Left Most Element"],
   Definition["Insert"], Definition["Concat"]⟩]
```

```
TS_Out[1532]=
⟨⟨⟨⟨ϵ, 3, ϵ⟩, 10, ϵ⟩, 10, ϵ⟩, 11, ϵ⟩, 100, ϵ⟩
```

Prove:

(Proposition (Problem of Sorting)) $\forall_X \exists_T (X \approx T \bigwedge \texttt{IsSorted}[T])$,

under the assumptions:

(Definition (IsSorted): 1) $\texttt{IsSorted}[\epsilon]$,

(Definition (IsSorted): 2) $\forall_{m,L,R} (\texttt{IsSorted}[L] \bigwedge \texttt{IsSorted}[R] \bigwedge \texttt{RgM}[L] \preceq m \bigwedge m \preceq \texttt{LfM}[R] \Rightarrow \texttt{IsSorted}[\langle L, m, R\rangle])$,

(Proposition (RgM empty)) $\forall_m (\texttt{RgM}[\epsilon] \preceq m)$,

(Proposition (LfM empty)) $\forall_m (m \preceq \texttt{LfM}[\epsilon])$,

(Proposition (1-1)) $\forall_{n,L,R} (\langle L, n, R\rangle \approx \texttt{Insertion}[n, \texttt{Merge}[L, R]])$,

(Proposition (2)) $\forall_{n,L} (\texttt{IsSorted}[L] \Rightarrow \texttt{IsSorted}[\texttt{Insertion}[n, L]])$,

(Proposition (3)) $\forall_{L,R} (\texttt{IsSorted}[L] \bigwedge \texttt{IsSorted}[R] \Rightarrow \texttt{IsSorted}[\texttt{Merge}[L, R]])$.

We prove (Proposition (Problem of Sorting)) by Induction on X.

1. Base case 1: We have to find witness such that:

(1) $\epsilon \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Apply IR4 and one obtains the substitution $\{T^* \to \epsilon\}$ and the new goal is:

(15) $\texttt{IsSorted}[\epsilon]$.

Our goal (15) is proved because it is identical to our assumption (Definition (IsSorted): 1) and we are done.

2. Base case 2: We have to find witness such that:

(2) $\langle \epsilon, n_0, \epsilon\rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Apply IR4 and one obtains the substitution $\{T^* \to \langle \epsilon, n_0, \epsilon\rangle\}$ and the new goal is:

(17) $\texttt{IsSorted}[\langle \epsilon, n_0, \epsilon\rangle]$.

In order to prove (17) by (Definition (IsSorted): 2) using substitution $\{L \to \epsilon, m \to n_0, R \to \epsilon\}$, it is sufficient to prove:

(18) $\texttt{IsSorted}[\epsilon] \bigwedge \texttt{IsSorted}[\epsilon] \bigwedge \texttt{RgM}[\epsilon] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\epsilon]$.

Using (Definition (IsSorted): 1)our goal(18) becomes:

(19) $\texttt{IsSorted}[\epsilon] \bigwedge \texttt{RgM}[\epsilon] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\epsilon]$.

Using (Definition (IsSorted): 1)our goal(19) becomes:

(20) $\texttt{RgM}[\in] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\in]$.

In order to prove (20), by (Proposition (RgM empty)) using substitution $\{m \to n_0\}$, it is sufficient to prove:

(21) $n_0 \preceq \texttt{LfM}[\in]$.

Goal (21) is proved because is an instance of universal assumption (Proposition (LfM empty)) so we are done.

3. Induction case 1: Let the induction hypothesis be:

(3) $L_0 \approx T_1$,

(4) $\texttt{IsSorted}[T_1]$,

and find witness such that:

(5) $\langle L_0, n_0, \in \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

We rewrite our goal (5) by using the assumption (3) and it is sufficient to prove:

(23) $\langle T_1, n_0, \in \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle T_1, n_0, \epsilon \rangle\}$ and the new goal is:

(26) $\texttt{IsSorted}[\langle T_1, n_0, \in \rangle]$.

We transform our goal (26) into proving:

(29) $\texttt{IsSorted}[T_1] \bigwedge \texttt{RgM}[T_1] \preceq n_0$.

Using (4)our goal(29) becomes:

(30) $\texttt{RgM}[T_1] \preceq n_0$.

When we reach a goal like (30) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle \epsilon, n_0, T_1 \rangle\}$ and the new goal is:

(27) $\texttt{IsSorted}[\langle \in, n_0, T_1 \rangle]$.

We transform our goal (27) into proving:

(31) $\texttt{IsSorted}[T_1] \bigwedge n_0 \preceq \texttt{LfM}[T_1]$.

Using (4)our goal(31) becomes:

(32) $n_0 \preceq \texttt{LfM}[T_1]$.

When we reach a goal like (32) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, T_1]\}$ and the new goal is:

(28) $\texttt{IsSorted}[\texttt{Insertion}[n_0, T_1]]$.

In order to prove (28) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_1\}$, it is sufficient to prove:

(33) $\texttt{IsSorted}[T_1]$.

Our goal (33) is proved because it is identical to our assumption (4) and we are done.

4. Induction Case 2: Let the induction hypothesis be:

(6) $R_0 \approx T_2$,

(7) $\texttt{IsSorted}[T_2]$,

and find witness such that:

(8) $\langle \epsilon, n_0, R_0 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

We rewrite our goal (8) by using the assumption (6) and it is sufficient to prove:

(34) $\langle \epsilon, n_0, T_2 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle \epsilon, n_0, T_2 \rangle\}$ and the new goal is:

(37) $\texttt{IsSorted}[\langle \epsilon, n_0, T_2 \rangle]$.

We transform our goal (37) into proving:

(40) $\texttt{IsSorted}[T_2] \bigwedge n_0 \leq \texttt{LfM}[T_2]$.

Using (7)our goal(40) becomes:

(41) $n_0 \leq \texttt{LfM}[T_2]$.

When we reach a goal like (41) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle T_2, n_0, \epsilon \rangle\}$ and the new goal is:

(38) $\texttt{IsSorted}[\langle T_2, n_0, \epsilon \rangle]$.

We transform our goal (38) into proving:

(42) $\texttt{IsSorted}[T_2] \bigwedge \texttt{RgM}[T_2] \leq n_0$.

Using (7)our goal(42) becomes:

(43) $\texttt{RgM}[T_2] \leq n_0$.

When we reach a goal like (43) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, T_2]\}$ and the new goal is:

(39) $\texttt{IsSorted}[\texttt{Insertion}[n_0, T_2]]$.

In order to prove (39) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_2\}$, it is sufficient to prove:

(44) `IsSorted[T2]` .

Our goal (44) is proved because it is identical to our assumption (7) and we are done.

5. Induction Case 3: Assume:

(9) $L_1 \approx T_3$,

(10) `IsSorted[T3]`,

(11) $R_1 \approx T_4$,

(12) `IsSorted[T4]`,

and find witness such that:

(13) $\langle L_1, n_0, R_1 \rangle \approx \mathrm{T}^* \bigwedge \mathtt{IsSorted}[\mathrm{T}^*]$

We rewrite our goal (13) by using both the assumptions (9) and (11) and it suffices to prove:

(45) $\langle T_3, n_0, T_4 \rangle \approx \mathrm{T}^* \bigwedge \mathtt{IsSorted}[\mathrm{T}^*]$.

In order to prove (45), by (Proposition (1-1)) using substitution
$\{L \to T_3, n \to n_0, R \to T_4, T^* \to \text{Insertion}[n_0, \text{Merge}[T_3, T_4]]\}$, it is sufficient to prove:

(49) `IsSorted[Insertion[n0, Merge[T3, T4]]]` .

The new assumption[s] is/[are]:

In order to prove (49) by (Proposition (2)) using substitution $\{n \to n_0, L \to \text{Merge}[T_3, T_4]\}$, it is sufficient to prove:

(51) `IsSorted[Merge[T3, T4]]` .

In order to prove (51) by (Proposition (3)) using substitution $\{L \to T_3, R \to T_4\}$, it is sufficient to prove:

(52) `IsSorted[T3]` $\bigwedge$ `IsSorted[T4]` .

Using (10)our goal(52) becomes:

(53) `IsSorted[T4]` .

Our goal (53) is proved because it is identical to our assumption (12) and we are done.

□

Prove:

(Proposition (Conjecture 1)) $\underset{n,R}{\forall}\left(\texttt{IsSorted}[R] \Rightarrow \underset{T}{\exists}\left(\langle \in, n, R\rangle \approx T \bigwedge \texttt{IsSorted}[T]\right)\right)$,

under the assumptions:

(Definition (IsSorted): 1) $\texttt{IsSorted}[\in]$,

(Definition (IsSorted): 2)
$\underset{m,L,R}{\forall}\left(\texttt{IsSorted}[L] \bigwedge \texttt{IsSorted}[R] \bigwedge \texttt{RgM}[L] \preceq m \bigwedge m \preceq \texttt{LfM}[R] \Rightarrow \texttt{IsSorted}[\langle L, m, R\rangle]\right)$,

(Proposition (1)) $\underset{n,L}{\forall}\left(\langle L, n, \in\rangle \approx \texttt{Insertion}[n, L]\right)$,

(Proposition (1-1)) $\underset{n,L,R}{\forall}\left(\langle L, n, R\rangle \approx \texttt{Insertion}[n, \texttt{Merge}[L, R]]\right)$,

(Proposition (2)) $\underset{n,L}{\forall}\left(\texttt{IsSorted}[L] \Rightarrow \texttt{IsSorted}[\texttt{Insertion}[n, L]]\right)$,

(Proposition (3)) $\underset{L,R}{\forall}\left(\texttt{IsSorted}[L] \bigwedge \texttt{IsSorted}[R] \Rightarrow \texttt{IsSorted}[\texttt{Merge}[L, R]]\right)$,

(Proposition (RgM empty)) $\underset{m}{\forall}\left(\texttt{RgM}[\in] \preceq m\right)$,

(Proposition (LfM empty)) $\underset{m}{\forall}\left(m \preceq \texttt{LfM}[\in]\right)$.

We prove (Proposition (Conjecture 1)) by Induction on R.

1. Base case: We have to find witness such that:

(1) $\texttt{IsSorted}[\in] \Rightarrow \langle \in, n_0, \in\rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

In order to prove (1), we assume:

(5) $\texttt{IsSorted}[\in]$,

and we have to prove:

(6) $\langle \in, n_0, \in\rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

Apply IR4 and one obtains the substitution $\{T^* \to \langle \epsilon, n_0, \epsilon\rangle\}$ and the new goal is:

(9) $\texttt{IsSorted}[\langle \in, n_0, \in\rangle]$.

In order to prove (9) by (Definition (IsSorted): 2) using substitution $\{L \to \epsilon, m \to n_0, R \to \epsilon\}$, it is sufficient to prove:

(10) $\texttt{IsSorted}[\in] \bigwedge \texttt{IsSorted}[\in] \bigwedge \texttt{RgM}[\in] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\in]$.

Using (5)our goal(10) becomes:

(11) $\texttt{IsSorted}[\in] \bigwedge \texttt{RgM}[\in] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\in]$.

Using (5)our goal(11) becomes:

(12) $\mathtt{RgM}[\in] \preceq n_0 \bigwedge n_0 \preceq \mathtt{LfM}[\in]$.

In order to prove (12), by (Proposition (RgM empty)) using substitution $\{m \to n_0\}$, it is sufficient to prove:

(13) $n_0 \preceq \mathtt{LfM}[\in]$.

Goal (13) is proved because is an instance of universal assumption (Proposition (LfM empty)) so we are done.

2. Induction step: Assume:

(2) $\mathtt{IsSorted}[L_0] \Rightarrow \langle \in, n_0, L_0 \rangle \approx T_1 \bigwedge \mathtt{IsSorted}[T_1]$,

(3) $\mathtt{IsSorted}[R_0] \Rightarrow \langle \in, n_0, R_0 \rangle \approx T_2 \bigwedge \mathtt{IsSorted}[T_2]$,

and find witness such that:

(4) $\mathtt{IsSorted}[\langle L_0, m_0, R_0 \rangle] \Rightarrow \langle \in, n_0, \langle L_0, m_0, R_0 \rangle \rangle \approx \mathtt{T}^* \bigwedge \mathtt{IsSorted}[\mathtt{T}^*]$

In order to prove (4), we assume:

(15) $\mathtt{IsSorted}[\langle L_0, m_0, R_0 \rangle]$,

and we have to prove:

(16) $\langle \in, n_0, \langle L_0, m_0, R_0 \rangle \rangle \approx \mathtt{T}^* \bigwedge \mathtt{IsSorted}[\mathtt{T}^*]$

We transform the assumption (15) into:

(17) $\mathtt{IsSorted}[L_0]$,

(18) $\mathtt{IsSorted}[R_0]$,

(19) $\mathtt{RgM}[L_0] \preceq m_0$,

(20) $m_0 \preceq \mathtt{LfM}[R_0]$.

We apply Modus Ponens on (17) and on (2) and the new assumption is:

(21) $\langle \in, n_0, L_0 \rangle \approx T_1 \bigwedge \mathtt{IsSorted}[T_1]$,

We apply Modus Ponens on (18) and on (3) and the new assumption is:

(22) $\langle \in, n_0, R_0 \rangle \approx T_2 \bigwedge \mathtt{IsSorted}[T_2]$,

We split the conjunction (21) into its individual conjuncts:

(23) $\langle \in, n_0, L_0 \rangle \approx T_1$,

(24) $\mathtt{IsSorted}[T_1]$,

We split the conjunction (22) into its individual conjuncts:

(25) $\langle \in, n_0, R_0 \rangle \approx T_2$,

(26) $\mathtt{IsSorted}[T_2]$,

From the existing assumption (19) the following assumption is generated:

(27) $L_0 \preceq m_0$.

From the existing assumption (20) the following assumption is generated:

(28) $m_0 \preceq R_0$.

Since our goal (16) matches both (23) and (25) we generate two alternatives:

Case 1: One obtains the substitution $\{T^* \to \langle T_1, m_0, R_0\rangle\}$ and the new goal is:

(29) $\texttt{IsSorted}[\langle T_1, m_0, R_0\rangle]$.

We transform our goal (29) into proving:

(31) $\texttt{IsSorted}[T_1] \bigwedge \texttt{IsSorted}[R_0] \bigwedge \texttt{RgM}[T_1] \preceq m_0 \bigwedge m_0 \preceq \texttt{LfM}[R_0]$.

Using (24)our goal(31) becomes:

(32) $\texttt{IsSorted}[R_0] \bigwedge \texttt{RgM}[T_1] \preceq m_0 \bigwedge m_0 \preceq \texttt{LfM}[R_0]$.

Using (18)our goal(32) becomes:

(33) $\texttt{RgM}[T_1] \preceq m_0 \bigwedge m_0 \preceq \texttt{LfM}[R_0]$.

Using (20)our goal(33) becomes:

(34) $\texttt{RgM}[T_1] \preceq m_0$.

By matching (23) and (19)our goal (34) becomes:

(35) $n_0 \preceq m_0 \bigwedge L_0 \preceq m_0$.

Using (27)our goal(35) becomes:

(36) $n_0 \preceq m_0$.

When we reach a goal like (36) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle L_0, m_0, T_2\rangle\}$ and the new goal is:

(30) $\texttt{IsSorted}[\langle L_0, m_0, T_2\rangle]$.

We transform our goal (30) into proving:

(37) $\texttt{IsSorted}[L_0] \bigwedge \texttt{IsSorted}[T_2] \bigwedge \texttt{RgM}[L_0] \preceq m_0 \bigwedge m_0 \preceq \texttt{LfM}[T_2]$.

Using (17)our goal(37) becomes:

(38) $\texttt{IsSorted}[T_2] \bigwedge \texttt{RgM}[L_0] \preceq m_0 \bigwedge m_0 \preceq \texttt{LfM}[T_2]$.

Using (26)our goal(38) becomes:

(39) $\texttt{RgM}[L_0] \preceq m_0 \bigwedge m_0 \preceq \texttt{LfM}[T_2]$.

Using (19)our goal(39) becomes:

(40) $m_0 \preceq \mathtt{LfM}[T_2]$.

By matching (25) and (20)our goal (40) becomes:

(41) $m_0 \preceq n_0 \bigwedge m_0 \preceq R_0$.

Using (28)our goal(41) becomes:

(42) $m_0 \preceq n_0$.

When we reach a goal like (42) it becomes the conditional assumption on this branch!

□

Prove:

(Proposition (Problem of Sorting)) $\forall_X \exists_T (X \approx T \wedge \texttt{IsSorted}[T])$,

under the assumptions:

(Definition (IsSorted): 1) $\texttt{IsSorted}[\epsilon]$,

(Definition (IsSorted): 2) $\forall_{m,L,R} (\texttt{IsSorted}[L] \wedge \texttt{IsSorted}[R] \wedge \texttt{RgM}[L] \leq m \wedge m \leq \texttt{LfM}[R] \Rightarrow \texttt{IsSorted}[\langle L, m, R \rangle])$,

(Proposition (1-1)) $\forall_{n,L,R} (\langle L, n, R \rangle \approx \texttt{Insertion}[n, \texttt{Merge}[L, R]])$,

(Proposition (2)) $\forall_{n,L} (\texttt{IsSorted}[L] \Rightarrow \texttt{IsSorted}[\texttt{Insertion}[n, L]])$,

(Proposition (3)) $\forall_{L,R} (\texttt{IsSorted}[L] \wedge \texttt{IsSorted}[R] \Rightarrow \texttt{IsSorted}[\texttt{Merge}[L, R]])$,

(Proposition (RgM empty)) $\forall_m (\texttt{RgM}[\epsilon] \leq m)$,

(Proposition (LfM empty)) $\forall_m (m \leq \texttt{LfM}[\epsilon])$.

We prove (Proposition (Problem of Sorting)) by Induction on X.

1. Base case 1: We have to find witness such that:

(1) $\epsilon \approx \texttt{T}^* \wedge \texttt{IsSorted}[\texttt{T}^*]$.

Apply IR4 and one obtains the substitution $\{T^* \to \epsilon\}$ and the new goal is:

(15) $\texttt{IsSorted}[\epsilon]$.

Our goal (15) is proved because it is identical to our assumption (Definition (IsSorted): 1) and we are done.

2. Base case 2: We have to find witness such that:

(2) $\langle \epsilon, n_0, \epsilon \rangle \approx \texttt{T}^* \wedge \texttt{IsSorted}[\texttt{T}^*]$.

Apply IR4 and one obtains the substitution $\{T^* \to \langle \epsilon, n_0, \epsilon \rangle\}$ and the new goal is:

(17) $\texttt{IsSorted}[\langle \epsilon, n_0, \epsilon \rangle]$.

In order to prove (17) by (Definition (IsSorted): 2) using substitution $\{L \to \epsilon, m \to n_0, R \to \epsilon\}$, it is sufficient to prove:

(18) $\texttt{IsSorted}[\epsilon] \wedge \texttt{IsSorted}[\epsilon] \wedge \texttt{RgM}[\epsilon] \leq n_0 \wedge n_0 \leq \texttt{LfM}[\epsilon]$.

Using (Definition (IsSorted): 1)our goal(18) becomes:

(19) $\texttt{IsSorted}[\epsilon] \wedge \texttt{RgM}[\epsilon] \leq n_0 \wedge n_0 \leq \texttt{LfM}[\epsilon]$.

Using (Definition (IsSorted): 1)our goal(19) becomes:

(20) $\mathtt{RgM}[\in] \preceq n_0 \bigwedge n_0 \preceq \mathtt{LfM}[\in]$.

In order to prove (20), by (Proposition (RgM empty)) using substitution $\{m \to n_0\}$, it is sufficient to prove:

(21) $n_0 \preceq \mathtt{LfM}[\in]$.

Goal (21) is proved because is an instance of universal assumption (Proposition (LfM empty)) so we are done.

3. Induction case 1: Let the induction hypothesis be:

(3) $L_0 \approx T_1$,

(4) $\mathtt{IsSorted}[T_1]$,

and find witness such that:

(5) $\langle L_0, n_0, \in \rangle \approx \mathtt{T}^* \bigwedge \mathtt{IsSorted}[\mathtt{T}^*]$

We rewrite our goal (5) by using the assumption (3) and it is sufficient to prove:

(23) $\langle T_1, n_0, \in \rangle \approx \mathtt{T}^* \bigwedge \mathtt{IsSorted}[\mathtt{T}^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle T_1, n_0, \epsilon \rangle\}$ and the new goal is:

(26) $\mathtt{IsSorted}[\langle T_1, n_0, \in \rangle]$.

We transform our goal (26) into proving:

(29) $\mathtt{IsSorted}[T_1] \bigwedge \mathtt{RgM}[T_1] \preceq n_0$.

Using (4)our goal(29) becomes:

(30) $\mathtt{RgM}[T_1] \preceq n_0$.

When we reach a goal like (30) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle \epsilon, n_0, T_1 \rangle\}$ and the new goal is:

(27) $\mathtt{IsSorted}[\langle \in, n_0, T_1 \rangle]$.

We transform our goal (27) into proving:

(31) $\mathtt{IsSorted}[T_1] \bigwedge n_0 \preceq \mathtt{LfM}[T_1]$.

Using (4)our goal(31) becomes:

(32) $n_0 \preceq \mathtt{LfM}[T_1]$.

When we reach a goal like (32) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, T_1]\}$ and the new goal is:

(28) $\mathtt{IsSorted}[\mathtt{Insertion}[n_0, T_1]]$.

In order to prove (28) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_1\}$, it is sufficient to prove:

(33) $\texttt{IsSorted}[T_1]$.

Our goal (33) is proved because it is identical to our assumption (4) and we are done.

4. Induction Case 2: Let the induction hypothesis be:

(6) $R_0 \approx T_2$,

(7) $\texttt{IsSorted}[T_2]$,

and find witness such that:

(8) $\langle \epsilon, n_0, R_0 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

We rewrite our goal (8) by using the assumption (6) and it is sufficient to prove:

(34) $\langle \epsilon, n_0, T_2 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle \epsilon, n_0, T_2 \rangle\}$ and the new goal is:

(37) $\texttt{IsSorted}[\langle \epsilon, n_0, T_2 \rangle]$.

We transform our goal (37) into proving:

(40) $\texttt{IsSorted}[T_2] \bigwedge n_0 \leq \texttt{LfM}[T_2]$.

Using (7)our goal(40) becomes:

(41) $n_0 \leq \texttt{LfM}[T_2]$.

When we reach a goal like (41) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle T_2, n_0, \epsilon \rangle\}$ and the new goal is:

(38) $\texttt{IsSorted}[\langle T_2, n_0, \epsilon \rangle]$.

We transform our goal (38) into proving:

(42) $\texttt{IsSorted}[T_2] \bigwedge \texttt{RgM}[T_2] \leq n_0$.

Using (7)our goal(42) becomes:

(43) $\texttt{RgM}[T_2] \leq n_0$.

When we reach a goal like (43) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, T_2]\}$ and the new goal is:

(39) $\texttt{IsSorted}[\texttt{Insertion}[n_0, T_2]]$.

In order to prove (39) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_2\}$, it is sufficient to prove:

(44) $\texttt{IsSorted}[T_2]$.

Our goal (44) is proved because it is identical to our assumption (7) and we are done.

5. Induction Case 3: Assume:

(9) $L_1 \approx T_3$,

(10) $\texttt{IsSorted}[T_3]$,

(11) $R_1 \approx T_4$,

(12) $\texttt{IsSorted}[T_4]$,

and find witness such that:

(13) $\langle L_1, n_0, R_1 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

We rewrite our goal (13) by using both the assumptions (9) and (11) and it suffices to prove:

(45) $\langle T_3, n_0, T_4 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle T_3, n_0, T_4 \rangle\}$ and the new goal is:

(49) $\texttt{IsSorted}[\langle T_3, n_0, T_4 \rangle]$.

We transform our goal (49) into proving:

(52) $\texttt{IsSorted}[T_3] \bigwedge \texttt{IsSorted}[T_4] \bigwedge \texttt{RgM}[T_3] \leq n_0 \bigwedge n_0 \leq \texttt{LfM}[T_4]$.

Using (10)our goal(52) becomes:

(53) $\texttt{IsSorted}[T_4] \bigwedge \texttt{RgM}[T_3] \leq n_0 \bigwedge n_0 \leq \texttt{LfM}[T_4]$.

Using (12)our goal(53) becomes:

(54) $\texttt{RgM}[T_3] \leq n_0 \bigwedge n_0 \leq \texttt{LfM}[T_4]$.

When we reach a goal like (54) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle T_4, n_0, T_3 \rangle\}$ and the new goal is:

(50) $\texttt{IsSorted}[\langle T_4, n_0, T_3 \rangle]$.

We transform our goal (50) into proving:

(55) $\texttt{IsSorted}[T_4] \bigwedge \texttt{IsSorted}[T_3] \bigwedge \texttt{RgM}[T_4] \leq n_0 \bigwedge n_0 \leq \texttt{LfM}[T_3]$.

Using (12)our goal(55) becomes:

(56) $\texttt{IsSorted}[T_3] \bigwedge \texttt{RgM}[T_4] \leq n_0 \bigwedge n_0 \leq \texttt{LfM}[T_3]$.

Using (10)our goal(56) becomes:

(57) $\texttt{RgM}[T_4] \leq n_0 \bigwedge n_0 \leq \texttt{LfM}[T_3]$.

When we reach a goal like (57) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, \text{Merge}[T_3, T_4]]\}$ and the new goal is:

(51) $\texttt{IsSorted}[\texttt{Insertion}[n_0, \texttt{Merge}[T_3, T_4]]]$.

In order to prove (51) by (Proposition (2)) using substitution $\{n \to n_0, L \to \text{Merge}[T_3, T_4]\}$, it is sufficient to prove:

(58) $\texttt{IsSorted}[\texttt{Merge}[T_3, T_4]]$.

In order to prove (58) by (Proposition (3)) using substitution $\{L \to T_3, R \to T_4\}$, it is sufficient to prove:

(59) $\texttt{IsSorted}[T_3] \bigwedge \texttt{IsSorted}[T_4]$.

Using (10)our goal(59) becomes:

(60) $\texttt{IsSorted}[T_4]$.

Our goal (60) is proved because it is identical to our assumption (12) and we are done.

□

Prove:

(Proposition (Problem of Sorting)) $\forall_X \exists_T (X \approx T \bigwedge \texttt{IsSorted}[T])$,

under the assumptions:

(Definition (IsSorted): 1) $\texttt{IsSorted}[\in]$,

(Definition (IsSorted): 2)
$\forall_{m,L,R} (\texttt{IsSorted}[L] \bigwedge \texttt{IsSorted}[R] \bigwedge \texttt{RgM}[L] \preceq m \bigwedge m \preceq \texttt{LfM}[R] \Rightarrow \texttt{IsSorted}[\langle L, m, R\rangle])$,

(Proposition (1-5)) $\forall_{n,L,R,A,B} (\langle L, n, \in\rangle \approx A \bigwedge R \approx B \Rightarrow \langle L, n, R\rangle \approx \texttt{Merge}[A, B])$,

(Proposition (2)) $\forall_{n,L} (\texttt{IsSorted}[L] \Rightarrow \texttt{IsSorted}[\texttt{Insertion}[n, L]])$,

(Proposition (3)) $\forall_{L,R} (\texttt{IsSorted}[L] \bigwedge \texttt{IsSorted}[R] \Rightarrow \texttt{IsSorted}[\texttt{Merge}[L, R]])$.

We prove (Proposition (Problem of Sorting)) by Induction on X.

1. Base case 1: We have to find witness such that:

(1) $\in \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Apply IR4 and one obtains the substitution $\{T^* \to \epsilon\}$ and the new goal is:

(11) $\texttt{IsSorted}[\in]$.

Our goal (11) is proved because it is identical to our assumption (Definition (IsSorted): 1) and we are done.

2. Base case 2: Assume:

(2) $L_0 \approx T_1$,

(3) $\texttt{IsSorted}[T_1]$,

and find witness such that:

(4) $\langle L_0, n_0, \in\rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

We rewrite our goal (4) by using the assumption (2) and it is sufficient to prove:

(12) $\langle T_1, n_0, \in\rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle T_1, n_0, \epsilon\rangle\}$ and the new goal is:

(15) $\texttt{IsSorted}[\langle T_1, n_0, \in\rangle]$.

We transform our goal (15) into proving:

(18) $\texttt{IsSorted}[T_1] \bigwedge \texttt{RgM}[T_1] \preceq n_0$.

Using (3)our goal(18) becomes:

(19) $\texttt{RgM}[T_1] \leq n_0$.

When we reach a goal like (19) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle \epsilon, n_0, T_1 \rangle\}$ and the new goal is:

(16) $\texttt{IsSorted}[\langle \epsilon, n_0, T_1 \rangle]$.

We transform our goal (16) into proving:

(20) $\texttt{IsSorted}[T_1] \bigwedge n_0 \leq \texttt{LfM}[T_1]$.

Using (3)our goal(20) becomes:

(21) $n_0 \leq \texttt{LfM}[T_1]$.

When we reach a goal like (21) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, T_1]\}$ and the new goal is:

(17) $\texttt{IsSorted}[\texttt{Insertion}[n_0, T_1]]$.

In order to prove (17) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_1\}$, it is sufficient to prove:

(22) $\texttt{IsSorted}[T_1]$.

Our goal (22) is proved because it is identical to our assumption (3) and we are done.

3. Induction step: Assume as induction hypotheses:

(5) $\langle L_1, n_0, \epsilon \rangle \approx T_2$,

(6) $\texttt{IsSorted}[T_2]$,

(7) $R_0 \approx T_3$,

(8) $\texttt{IsSorted}[T_3]$,

and find witness such that:

(9) $\langle L_1, n_0, R_0 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

In order to prove (9) by (Proposition (1-5)) using substitution $\left\{L \to L_1, n \to n_0, R \to R_0, T^* \to \text{Merge}\left[A^{**}, B^{**}\right]\right\}$, it is sufficient to prove:

(26) $\langle L_1, n_0, \epsilon \rangle \approx \texttt{A}^{**} \bigwedge R_0 \approx \texttt{B}^{**} \bigwedge \texttt{IsSorted}[\texttt{Merge}[\texttt{A}^{**}, \texttt{B}^{**}]]$.

In order to prove (26) by (5) using substitution $\left\{A^{**} \to T_2\right\}$, it is sufficient to prove:

(27) $R_0 \approx \texttt{B}^{**} \bigwedge \texttt{IsSorted}[\texttt{Merge}[T_2, \texttt{B}^{**}]]$.

In order to prove (27) by (7) using substitution $\left\{B^{**} \to T_3\right\}$, it is sufficient to prove:

(28) `IsSorted[Merge[T2, T3]]` .

In order to prove (28) by (Proposition (3)) using substitution $\{L \to T_2, R \to T_3\}$, it is sufficient to prove:

(29) `IsSorted[T2]` $\wedge$ `IsSorted[T3]` .

Using (6)our goal(29) becomes:

(30) `IsSorted[T3]` .

Our goal (30) is proved because it is identical to our assumption (8) and we are done.

□

Prove:

(Proposition (Problem of Sorting)) $\forall_X \exists_T (X \approx T \bigwedge \texttt{IsSorted}[T])$,

under the assumptions:

(Definition (IsSorted): 1) $\texttt{IsSorted}[\epsilon]$,

(Definition (IsSorted): 2)

$\forall_{m,L,R} (\texttt{IsSorted}[L] \bigwedge \texttt{IsSorted}[R] \bigwedge \texttt{RgM}[L] \preceq m \bigwedge m \preceq \texttt{LfM}[R] \Rightarrow \texttt{IsSorted}[\langle L, m, R\rangle])$,

(Proposition (RgM empty)) $\forall_m (\texttt{RgM}[\epsilon] \preceq m)$,

(Proposition (LfM empty)) $\forall_m (m \preceq \texttt{LfM}[\epsilon])$,

(Proposition (2)) $\forall_{n,L} (\texttt{IsSorted}[L] \Rightarrow \texttt{IsSorted}[\texttt{Insertion}[n, L]])$,

(Proposition (1-6))

$\forall_{n,L,R,A,B,C} (L \approx A \bigwedge R \approx B \bigwedge \texttt{Concat}[L, R] \approx C \Rightarrow \langle A, n, B\rangle \approx \texttt{Insertion}[n, C])$.

We prove (Proposition (Problem of Sorting)) by Induction on X.

1. Base case 1: We have to find witness such that:

(1) $\epsilon \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Apply IR4 and one obtains the substitution $\{T^* \to \epsilon\}$ and the new goal is:

(15) $\texttt{IsSorted}[\epsilon]$.

Our goal (15) is proved because it is identical to our assumption (Definition (IsSorted): 1) and we are done.

2. Base case 2: We have to find witness such that:

(2) $\langle \epsilon, n_0, \epsilon\rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

Apply IR4 and one obtains the substitution $\{T^* \to \langle \epsilon, n_0, \epsilon\rangle\}$ and the new goal is:

(17) $\texttt{IsSorted}[\langle \epsilon, n_0, \epsilon\rangle]$.

In order to prove (17) by (Definition (IsSorted): 2) using substitution $\{L \to \epsilon, m \to n_0, R \to \epsilon\}$, it is sufficient to prove:

(18) $\texttt{IsSorted}[\epsilon] \bigwedge \texttt{IsSorted}[\epsilon] \bigwedge \texttt{RgM}[\epsilon] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\epsilon]$.

Using (Definition (IsSorted): 1)our goal(18) becomes:

(19) $\texttt{IsSorted}[\epsilon] \bigwedge \texttt{RgM}[\epsilon] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\epsilon]$.

Using (Definition (IsSorted): 1)our goal(19) becomes:

(20) $\texttt{RgM}[\epsilon] \preceq n_0 \bigwedge n_0 \preceq \texttt{LfM}[\epsilon]$.

In order to prove (20), by (Proposition (RgM empty)) using substitution $\{m \to n_0\}$, it is sufficient to prove:

(21) $n_0 \leq \mathtt{LfM}[\in]$.

Goal (21) is proved because is an instance of universal assumption (Proposition (LfM empty)) so we are done.

3. Induction case 1: Let the induction hypothesis be:

(3) $L_0 \approx T_1$,

(4) $\mathtt{IsSorted}[T_1]$,

and find witness such that:

(5) $\langle L_0, n_0, \in \rangle \approx \mathtt{T}^* \bigwedge \mathtt{IsSorted}[\mathtt{T}^*]$

We rewrite our goal (5) by using the assumption (3) and it is sufficient to prove:

(23) $\langle T_1, n_0, \in \rangle \approx \mathtt{T}^* \bigwedge \mathtt{IsSorted}[\mathtt{T}^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle T_1, n_0, \epsilon \rangle\}$ and the new goal is:

(26) $\mathtt{IsSorted}[\langle T_1, n_0, \in \rangle]$.

We transform our goal (26) into proving:

(29) $\mathtt{IsSorted}[T_1] \bigwedge \mathtt{RgM}[T_1] \leq n_0$.

Using (4)our goal(29) becomes:

(30) $\mathtt{RgM}[T_1] \leq n_0$.

When we reach a goal like (30) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle \epsilon, n_0, T_1 \rangle\}$ and the new goal is:

(27) $\mathtt{IsSorted}[\langle \in, n_0, T_1 \rangle]$.

We transform our goal (27) into proving:

(31) $\mathtt{IsSorted}[T_1] \bigwedge n_0 \leq \mathtt{LfM}[T_1]$.

Using (4)our goal(31) becomes:

(32) $n_0 \leq \mathtt{LfM}[T_1]$.

When we reach a goal like (32) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, T_1]\}$ and the new goal is:

(28) $\mathtt{IsSorted}[\mathtt{Insertion}[n_0, T_1]]$.

In order to prove (28) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_1\}$, it is sufficient to prove:

(33) `IsSorted[T1]` .

Our goal (33) is proved because it is identical to our assumption (4) and we are done.

4. Induction Case 2: Let the induction hypothesis be:

(6) $R_0 \approx T_2$,

(7) `IsSorted[T2]`,

and find witness such that:

(8) $\langle \in, n_0, R_0 \rangle \approx T^* \bigwedge \text{IsSorted}[T^*]$

We rewrite our goal (8) by using the assumption (6) and it is sufficient to prove:

(34) $\langle \in, n_0, T_2 \rangle \approx T^* \bigwedge \text{IsSorted}[T^*]$.

Generate the corresponding permutations and prove by cases:

Case 1: One obtains the substitution $\{T^* \to \langle \epsilon, n_0, T_2 \rangle\}$ and the new goal is:

(37) `IsSorted[`$\langle \in, n_0, T_2 \rangle$`]` .

We transform our goal (37) into proving:

(40) `IsSorted[T2]` $\bigwedge n_0 \leq$ `LfM[T2]` .

Using (7)our goal(40) becomes:

(41) $n_0 \leq$ `LfM[T2]` .

When we reach a goal like (41) it becomes the conditional assumption on this branch!

Case 2: One obtains the substitution $\{T^* \to \langle T_2, n_0, \epsilon \rangle\}$ and the new goal is:

(38) `IsSorted[`$\langle T_2, n_0, \in \rangle$`]` .

We transform our goal (38) into proving:

(42) `IsSorted[T2]` $\bigwedge$ `RgM[T2]` $\leq n_0$.

Using (7)our goal(42) becomes:

(43) `RgM[T2]` $\leq n_0$.

When we reach a goal like (43) it becomes the conditional assumption on this branch!

Case 3: One obtains the substitution $\{T^* \to \text{Insertion}[n_0, T_2]\}$ and the new goal is:

(39) `IsSorted[Insertion[n0, T2]]` .

In order to prove (39) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_2\}$, it is sufficient to prove:

(44) `IsSorted[T2]` .

Our goal (44) is proved because it is identical to our assumption (7) and we are done.

5. Induction Case 3: Assume:

(9) $L_1 \approx T_3$,

(10) $\texttt{IsSorted}[T_3]$,

(11) $R_1 \approx T_4$,

(12) $\texttt{IsSorted}[T_4]$,

and find witness such that:

(13) $\langle L_1, n_0, R_1 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$

From the existing assumptions (9) (10) (11) and (12) the following assumptions are generated:

(45) $\texttt{Concat}[L_1, R_1] \approx T_5$,

(46) $\texttt{IsSorted}[T_5]$.

We rewrite our goal (13) by using both the assumptions (9) and (11) and it suffices to prove:

(47) $\langle T_3, n_0, T_4 \rangle \approx \texttt{T}^* \bigwedge \texttt{IsSorted}[\texttt{T}^*]$.

In order to prove (47) by (Proposition (1-6)) using substitution $\{A \to T_3, n \to n_0, B \to T_4, T^* \to \text{Insertion}[n_0, C^{**}]\}$, it is sufficient to prove:

(52) $\texttt{L}^{**} \approx T_3 \bigwedge \texttt{R}^{**} \approx T_4 \bigwedge \texttt{Concat}[\texttt{L}^{**}, \texttt{R}^{**}] \approx \texttt{C}^{**} \bigwedge \texttt{IsSorted}[\texttt{Insertion}[n_0, \texttt{C}^{**}]]$.

In order to prove (52) by (9) using substitution $\{L^* \to L_1\}$, it is sufficient to prove:

(53) $\texttt{R}^{**} \approx T_4 \bigwedge \texttt{Concat}[L_1, \texttt{R}^{**}] \approx \texttt{C}^{**} \bigwedge \texttt{IsSorted}[\texttt{Insertion}[n_0, \texttt{C}^{**}]]$.

In order to prove (53) by (11) using substitution $\{R^* \to R_1\}$, it is sufficient to prove:

(54) $\texttt{Concat}[L_1, R_1] \approx \texttt{C}^{**} \bigwedge \texttt{IsSorted}[\texttt{Insertion}[n_0, \texttt{C}^{**}]]$.

In order to prove (54) by (45) using substitution $\{C^{**} \to T_5\}$, it is sufficient to prove:

(55) $\texttt{IsSorted}[\texttt{Insertion}[n_0, T_5]]$.

In order to prove (55) by (Proposition (2)) using substitution $\{n \to n_0, L \to T_5\}$, it is sufficient to prove:

(56) $\texttt{IsSorted}[T_5]$.

Our goal (56) is proved because it is identical to our assumption (46) and we are done.

□

6 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 proofs, an extractor in the *Theorema* system which is able to extract from a proof the corresponding algorithms (including `if-then-else` algorithms), the synthesis of numerous sorting algorithms and auxiliary algorithms. We have also certified by *Coq* the soundness property of F_1 with the current implementation of the auxiliary functions. The certification proof is more complex and its generation less automatic than for the *Theorema* proof that helped for extracting F_1, by using different inference rules and additional properties.

The problem of sorting binary trees does not appear to have an important practical significance, and in fact the algorithms we synthesize are not very efficient. (For instance it appears to be more efficient to extract the elements of the tree in a list, to sort it by a fast algorithm, and then to construct the sorted tree.) However, the problem itself poses interesting algorithmic problems, and also the proof techniques are more involved than the ones from lists. This is very relevant for our research, because our primary goal is not to generate the most efficient algorithms, but to study interesting examples of proving and synthesis, from which we can discover *new proof methods for algorithm synthesis*.

Our experiments done in the *Theorema* system show that by applying different induction principles and by choosing different alternatives in the proofs one can discover numerous algorithms for the same functions, differing in efficiency and complexity. This case study illustrates that the automation of the synthesis problem is not a trivial one.

As further work, for a fully automatization of the synthesis process, we want to use other systems in order to automatically generate the induction principles, which in the *Theorema* system are given as inference rules in the prover. We also want to use the method presented in this paper on more complex recursive data structures (e.g. red-black trees). In the near future, we intend to certify the correctness property for the other synthesized sorting algorithms. One of our long-term goals is to define procedures for translating the *Theorema* proofs directly into Coq scripts, by following similar translation procedures as those used for implicit induction proofs [14, 22].

Acknowledgements

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

References

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

2. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer Verlag, 1998.
3. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science. An EATCS*. Springer, 2004.
4. B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. In *Analele Universitatii de Vest, Timisoara, Ser. Matematica - Informatica*, volume XLI, pages 41–70, 2003.
5. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.
6. A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing Induction Rules for Deductive Synthesis Proofs. *Electron. Notes Theor. Comput. Sci.*, 153:3–21, March 2006.
7. C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer International Publishing, 2013.
8. B. Delaware, C. P. Claudel, J. Gross, and A. Chlipala. 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, pages 689–700, New York, NY, USA, 2015. ACM.
9. I. Dramnesc and T. Jebelean. Proof Techniques for Synthesis of Sorting Algorithms. In *Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, number ISBN 978-0-7695-4630-8, pages 101–109. IEEE Computer Society, September 2011.
10. I. Dramnesc and T. Jebelean. Discovery of Inductive Algorithms through Automated Reasoning: A Case Study on Sorting. In *Proceedings of IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*, number ISBN 978-1-4673-4751-8, pages 293 – 298. IEEE Xplore, September 2012.
11. I. Dramnesc and T. Jebelean. Theory Exploration in Theorema: Case Study on Lists. In *Proceedings of IEEE 7th International Symposium on Applied Computational Intelligence and Informatics*, number ISBN 978-1-4673-1013-0, pages 421–426. IEEE Xplore, May 2012.
12. I. Dramnesc and T. Jebelean. Synthesis of list algorithms by mechanical proving. *Journal of Symbolic Computation*, 68:61–92, 2015.
13. S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM.
14. A. Henaien and S. Stratulat. Performing implicit induction reasoning with certifying proof environments. In A. Bouhoula, T. Ida, and F. Kamareddine, editors, *Proceedings Fourth International Symposium on Symbolic Computation in Software Science, Gammarth, Tunisia, 15-17 December 2012*, volume 122 of *Electronic Proceedings in Theoretical Computer Science*, pages 97–108. Open Publishing Association, 2013.
15. E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 407–426, New York, NY, USA, 2013. ACM.
16. D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
17. R. Kowalski and D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2, 1971.
18. B. Mordechai. *Mathematical Logic for Computer Science*. Springer, 2004.

19. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
20. D. R. Smith. Generating programs plus proofs by refinement. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 182–188. Springer, 2005.
21. S. Stratulat. A Unified View of Induction Reasoning for First-Order Logic. In A. Voronkov, editor, *Turing-100 (The Alan Turing Centenary Conference)*, volume 10 of *EPiC Series*, pages 326–352. EasyChair, 2012.
22. S. Stratulat and V. Demange. Automated certification of implicit induction proofs. In *CPP'2011 (First International Conference on Certified Programs and Proofs)*, volume 7086 of *Lecture Notes Computer Science*, pages 37–53. Springer Verlag, 2011.
23. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, April 1971.
24. S. Wolfram. *The Mathematica Book*. Wolfram Media Inc., 2003.