Proof–based Synthesis of Sorting Algorithms for Trees

Isabela Dramnesc, Tudor Jebelean, Sorin Stratulat

Proof–based Synthesis of Sorting Algorithms for Trees

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.

Keywords: algorithm synthesis, sorting, theorem proving

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 $O[X,T]$, where X and T are vectors of universal and existential variables, respectively. The desired function F must satisfy the correctness condition $(\forall X)(I[X] \Longrightarrow O[X, F[X]])$.[1]

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 [5]. 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

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

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.

Non-trivial algorithms, as for sorting [14], 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 [8]. 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. For lack of space, we fully present the synthesis process for one of these algorithms; the proofs for the other algorithms are only summarized but can be found in the technical report [9]. 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[2] and e.g., [4] which is itself implemented in Mathematica [20]. *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 can be accessed in the technical report [9].

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 [12]. Synthesis methods and techniques similar to our proof-based approach are extensively presented in [8]. 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, 19] integrate techniques that are mainly executed manually and implemented by higher-order proof assistants like Isabelle/HOL [15] or more synthesis-oriented tools as Specware [16]. 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 [13]. 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 [7] to ensure the soundness of the validation step by certification with the Coq kernel. [6] 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

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

implemented) version. Implementing inference rules directly in Coq may be of interest if one can prove that every generated synthesized algorithm is sound. In general, this is a rather difficult task, therefore this approach does not fit for rapid prototyping and testing new ideas.

2 The Proof-based Synthesis Method

This section introduces the algorithm synthesis problem and presents the proof-based synthesis techniques that we use, by adapting and improving inference rules and induction principles from [8].

2.1 Our Approach

Basic notions and notations. According 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 appear under the quantifier: $\underset{X}{\forall}$ ("for all X") and $\underset{T}{\exists}$ ("exists T"). We consider binary trees over a totally ordered domain. In our formulae there are two kinds of objects: domain objects which are tree members (usually denoted by lower-case letters – e.g. a, b, n), and binary trees (usually represented by upper-case letters – e.g. X, T, Y, Z). However the formulae do not indicate explicitly the types of the objects, but our specific predicate and function symbols are not overloaded[3]. Furthermore the meta–variables are starred (e.g., T^*, T_1^*, Z^*) and the Skolem constants have integer indices (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, i.e., the rightmost (resp. leftmost) element; $Concat[X, Y]$ 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:

[3] Each predicate and function symbol applies to a certain combination of types of argument.

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

An example of simple property which can be proven inductively from Definition 3 is the following:

Property 5. $\underset{z,T}{\forall}\Big(IsSorted[T]\Longrightarrow(T\preceq z\Longleftrightarrow RgM[T]\leq z)\Big)$

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: $\underset{XT}{\forall\exists}(I[X]\Rightarrow O[X,T])$. 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 6. $\underset{X}{\forall}\underset{T}{\exists}(X\approx T\wedge IsSorted[T])$

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. An example of knowledge base is given in [10]. 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). Some of their necessary properties are:

Proposition 7. $\underset{T}{\forall}\Big(IsSorted[T] \Longrightarrow IsSorted[Insert[n, T]]\Big)$

Proposition 8. $\underset{L,R}{\forall}\Big((IsSorted[L] \wedge IsSorted[R]) \Longrightarrow IsSorted[Merge[L, R]]\Big)$

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

The illustration of the induction principles and algorithm extraction in this subsection is similar to the one from [8], 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 [17] 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 used different induction principles for proving P as unary predicate over binary trees. A first example is:

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.

Induction-2:

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

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.
Induction schema discovery. In some examples (e.g., synthesis of $Merge[X, Y]$ [11]), the induction principles are generated in a lazy way, especially when 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

We summarize here the main inference rules and proof strategies. More details are given in [9].
Inference rules:

IR-1: *Generate Microatoms.* Certain atoms can be transformed into a conjunction of (micro)atoms, depending on the specific properties of our functions and predicates. E.g., $IsSorted[\langle T_1, n, T_2\rangle]$ is transformed into $(IsSorted[T_1] \wedge IsSorted[T_2] \wedge RgM[T_1] \leq n \wedge n \leq LfM[T_2])$. Similarly, we get $x \preceq A \ \wedge x \leq b \wedge \ x \preceq C$ from $x \preceq \langle A, b, C\rangle$.

IR-2: *Eliminate-Ground-Formulae-from-Goal.* The ground formulas from any goal are deleted if they are assumption instances.

IR-3: *Replace-Equivalent-Term-in-Goal.* Let $t_1 \approx t_2$ be an assumption and assume that t_1 occurs in a goal as argument of a predicate which is preserved by equivalence $(\approx, \preceq)$. The rule replaces t_1 by t_2.

IR-4: *Generate permutations and expressions.* This rule applies combinatorial techniques that are widely explained in [11]. Given a goal of the form $Expression \approx T^* \wedge IsSorted[T^*]$, it generates all *permutations* of the list of nonempty symbols from *Expression*. Then, for each permutation it generates all possible witnesses as a tree *expressions* containing these symbols. E.g., if *Expression* is $\langle L, x, \varepsilon\rangle$, then the generated trees are: $\langle L, x, \varepsilon\rangle$, $\langle \varepsilon, x, L\rangle$, and $Insert[x, L]$.

Strategies:

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

S-2: *Priority-of-Local-Assumptions.* The strategy consists in using with priority the local assumptions, usually ground formulae generated during the current proof and considered as "true" in the context of the proof, w.r.t. the global assumptions consisting of definitions and propositions from the database, considered as being always "true".

3 Experiments

3.1 Synthesis of Sort-1

In this subsection we present the automatically generated proof of Conjecture 6 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 6 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:

$$\underset{T}{\exists}(L_0 \approx T \wedge \mathit{IsSorted}[T]) \tag{1}$$

and prove:

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

Apply **S-1** on (1) and (2) to eliminate the existential quantifiers. The induction hypotheses are:

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

and the goal is:

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

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

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

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

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

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

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

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

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

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

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

which becomes the conditional assumption on this branch.
Alternative-3: Since the disjunction of the conditions (8) and (9) is not provable, the prover generates a further alternative. This depends on the synthesis scenario (see the end of Section 2.1). If the properties of the function *Insert* are present in the knowledge base, then the prover generates the substitution $\{T^* \to Insert[n, T_1]\}$ based on these properties.

If the properties of *Insert* are not present, then the prover generates a failing branch. A new conjecture is further generated which is used for the synthesis of *Insert*. 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: $\{T^* \to \langle \varepsilon, n, T_2 \rangle\}$ and the conditional assumption is: $n \leq LfM[T_2]$.
Alternative-2: $\{T^* \to \langle T_2, n, \varepsilon \rangle\}$ 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: $\{T^* \to Insert[n, T_2]\}$.

Induction case 3: Assume:

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

and prove:

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

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

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

Apply **IR-4** and 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.

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. 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, knowing that if the proof succeeds to find a witness $T^* = \Im[n, L_0, R_0, T_1, T_2]$ (term depending

on n, L_0, R_0, T_1 and T_2), then a new branch $F[\langle L, n, R\rangle] = \Im[n, L, R, F[L], F[R]]$ of the synthesized algorithm is generated (T_1 and T_2 are replaced by $F[L]$ and $F[R]$, respectively) by using as conditions the conditional assumptions required by the witness:

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

3.2 Additional Certification of the Synthesized Algorithm F_1

The theoretical basis and the correctness of this proof-based synthesis scheme is well known – see for instance [5]. However, 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 9. $\underset{n,m,L,R,S}{\forall} \left(\begin{array}{c} RgM[L, n, \varepsilon] = n \\ RgM[L, n, \langle R, m, S\rangle] = RgM[R, m, S] \end{array} \right)$

Definition 10. $\underset{n,m,L,R,S}{\forall} \left(\begin{array}{c} LfM[\varepsilon, n, R] = n \\ LfM[\langle L, n, R\rangle, m, S] = LfM[L, n, R] \end{array} \right)$

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.[4]

3.3 Synthesis of Other Sorting Algorithms

Sort-2. The prover generated automatically the proof of Conjecture 6 by applying **Induction-2** and by using the current knowledge base (Definition 4, Proposition 7, and 8), including the following property:

Proposition 11. $\underset{n,L,R,A,B}{\forall} ((\langle L, n, \varepsilon\rangle \approx A \wedge R \approx B) \Longrightarrow \langle L, n, R\rangle \approx Merge[A, B])$

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

[4] The full Coq script is available at: `http://web.info.uvt.ro/~idramnesc/LATA2016/coq.v`

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

Sort-3. The proof of Conjecture 6 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 6 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 6 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:

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

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 technical report [9].

The following table presents the synthesized sorting algorithms. For each of them Conjecture 6 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

4 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 and presented in detail in the technical report [9] 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. For example, we can apply induction schemas that are issued from recursive data structures and functions defined in Coq [18]. 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, using a similar approach as for F_1. 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 [18].

Acknowledgments

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

2. Back, R.J., von Wright, J.: Refinement Calculus. Springer Verlag (1998)
3. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science. An EATCS, vol. XXV. Springer (2004)
4. Buchberger, B., Craciun, A., Jebelean, T., Kovacs, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: Towards computer-aided mathematical theory exploration. Journal of Applied Logic 4(4), 470–504 (2006)
5. Bundy, A., Dixon, L., Gow, J., Fleuriot, J.: Constructing induction rules for deductive synthesis proofs. Electron. Notes Theor. Comput. Sci. 153, 3–21 (March 2006)
6. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: Gonthier, G., Norrish, M. (eds.) Certified Programs and Proofs, Lecture Notes in Computer Science, vol. 8307, pp. 147–162. Springer International Publishing (2013)
7. Delaware, B., Claudel, C.P., Gross, J., Chlipala, A.: 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. pp. 689–700. POPL '15, ACM, New York, NY, USA (2015)
8. Dramnesc, I., Jebelean, T.: Synthesis of list algorithms by mechanical proving. Journal of Symbolic Computation 68, 61–92 (2015)
9. Dramnesc, I., Jebelean, T., Stratulat, S.: Synthesis of some algorithms for trees: Experiments in Theorema. Tech. Rep. 15-04, RISC Report Series, Johannes Kepler University, Linz, Austria (2015)
10. Dramnesc, I., Jebelean, T., Stratulat, S.: Theory exploration of binary trees. In: 13th IEEE International Symposium on Intelligent Systems and Informatics (SISY 2015). pp. 139–144. IEEE Publishing (2015)
11. Dramnesc, I., Jebelean, T., Stratulat, S.: Combinatorial techniques for proof-based synthesis of sorting algorithms. In: SYNASC 2015: Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (to appear)
12. Gulwani, S.: Dimensions in program synthesis. In: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming. pp. 13–24. PPDP '10, ACM, New York, NY, USA (2010)
13. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. pp. 407–426. OOPSLA '13, ACM, New York, NY, USA (2013)
14. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley Longman Publishing, Redwood City, CA, USA, 2 edn. (1998)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
16. Smith, D.R.: Generating programs plus proofs by refinement. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 182–188. Springer (2005)
17. Stratulat, S.: A unified view of induction reasoning for first-order logic. In: Voronkov, A. (ed.) Turing-100 (The Alan Turing Centenary Conference). EPiC Series, vol. 10, pp. 326–352. EasyChair (2012)
18. Stratulat, S.: Mechanically certifying formula-based noetherian induction reasoning. Journal of Symbolic Computation (accepted), `http:lita.univ-lorraine.fr/~stratula/jsc2016.pdf`
19. Wirth, N.: Program development by stepwise refinement. Commun. ACM 14(4), 221–227 (Apr 1971)
20. Wolfram, S.: The Mathematica Book. Wolfram Media Inc. (2003)