

Synthesis of Algorithms on Sets Represented as Monotone Lists

Isabela Drămnesc *

Department of Computer Science, West University,
V.Parvan 4, 300223, Timișoara, Romania
`idramnesc@info.uvt.ro`

Extended abstract.

We start from the set theory axioms which are described in [2], see also [5]. We choose to represent sets as sorted lists without duplications, which we call *monotone* lists. Without loss of generality, we assume that the elements of the sets are from an ordered domain, and we take increasingly sorted lists. This approach leads to more efficient algorithms than the ones for operating on sets represented as non-sorted lists. We use two functions: the function R which applied to a set returns its representation as a monotone list, and the function S which applied to a (monotone) list returns its corresponding set. These two functions are bijective and reverse to each other and represent the isomorphism between the domain of finite sets (with the usual signature) and the domain of monotone lists with the appropriate signature induced by this isomorphism. The synthesis problem consists in producing the algorithms implementing this signature.

We use the *Theorema* system, (www.theorema.org and e.g. [3]) which is based on Mathematica - see [6].

Notations.

We use lower-case roman letters for elements (e.g. a, b) and upper-case roman letters for lists (e.g. A, B, C) and for sets (e.g. X, Y, Z). We do not use explicitly the types of objects, but we use predicate and function symbols which are not overloaded. In line with the tradition of the *Theorema* system we use square brackets for predicate and for function application (e. g. $P[x], f[x]$) instead of the usual $P(x), f(x)$.

Lists: The construction of lists is made by using: $\langle \rangle$ (for the empty list) and $\smile$ (which adds an element at the beginning of a list). Examples: $a \smile \langle \rangle$ is the list containing one single element and $a \smile A$ is the list with the first element a and the rest (tail) of the list A .

Quantifiers: $\forall_X$ ("for all X ") denotes that X is a universally quantified variable; $\exists_T$ ("exists T ") denotes that T is an existentially quantified variable;

* The work reported in this article was partially supported by the European Commission grant no. FP7-REGPOT-CT-2011-284595 (HOST).

We consider strictly ordering relations. E.g., $(a < b)$ denotes that the element a is strictly smaller than the element b , $(a \prec A)$ denotes that a is strictly smaller than all the elements from the list A , and $(A \ll B)$ denotes that all the elements from A are strictly smaller than all the elements from B .

Sets: We employ the usual notation: $\{\}$ (the empty set), $\{a\}$ is a set containing one single element, the predicate symbol $\in$ for member, the predicate symbol $\subseteq$ for non-strict inclusion, the function $\cup$ for the union, the function $\cap$ for the intersection and the function $\setminus$ for the difference between two sets.

Representation of sets: We choose to represent sets as monotone lists. We use the two functions R and S . R is the function which applied to a set returns its representation, namely monotone lists, e.g. $R[\{2, 1, 4\}]$ returns the list $(1 \smile (2 \smile (4 \smile \langle \rangle)))$ and S is the function which applied to a list returns its corresponding set, e.g. $S[1 \smile (2 \smile (3 \smile (4 \smile \langle \rangle)))]$ returns the set $\{1, 2, 3, 4\}$. Attention that S is always applied to monotone lists only.

Proof Based Synthesis

The Synthesis Problem. In this context we want to synthesize different algorithms (predicates and functions) for sets represented as monotone lists.

Input: G – a function on sets.

Output: The algorithm implementing the isomorphic function F on monotone lists. (The algorithm is a set of conditional equalities which define F .) For instance, if G (thus also F) are binary functions, then F must satisfy:

Case 1: Function synthesis the problem is:

Proposition 1. $\forall_{A,B} (S[F[A, B]] = G[S[A], S[B]])$

Here A, B are arbitrary monotone lists, F is the function which we have to discover (synthesize) and G is a function applied to sets for which we have the defining properties in the knowledge base.

The synthesis problem for predicates is similar, but the algorithm will be expressed as a set of conditional equivalences. For instance in the case of binary predicates we have:

Case 2: Predicate synthesis the problem is:

Proposition 2. $\forall_{A,B} (P[A, B] \Leftrightarrow W[S[A], S[B]])$

Here A, B are arbitrary monotone lists, P is the predicate which we have to synthesize, and W is a predicate on sets for which we have the defining properties in the knowledge base.

Synthesis Method: Inductive Proof. Each synthesis starts as an inductive constructive proof which applies specific strategies and inference rules. These

strategies and inference rules are based on the corresponding properties of sets (as monotone lists). We synthesize (by extracting from proofs) five algorithms among with two predicates: **Member** and **Inclusion**; and three functions: **Union**, **Intersection**, and **Difference**.

First start to prove Proposition 1 or Proposition 2 by applying one of the following induction principles:

Head-Tail-Induction-1:

$$\left(Q[\langle \rangle] \wedge \forall_{a,X} (Q[X] \Rightarrow Q[a \smile X]) \right) \Rightarrow \forall_X Q[X]$$

Head-Tail-Double-Induction

$$\begin{aligned} & \left(Q[\langle \rangle, \langle \rangle] \wedge \forall_A (Q[A, \langle \rangle]) \wedge \forall_B (Q[\langle \rangle, B]) \wedge \right. \\ & \left. \forall_{a,b,A,B} ((Q[A, B] \wedge Q[a \smile A, B] \wedge Q[A, b \smile B]) \Rightarrow Q[a \smile A, b \smile B]) \right) \\ & \Rightarrow \forall_{A,B} Q[A, B] \end{aligned}$$

During the proof we add the necessary properties in the knowledge base and by applying specific strategies and inference rules we obtain a constructive proof. If the proof succeeds, then extract the algorithm from the proof. Of course there may be different proofs of the same fact. In this chapter we only present one version of each proof, but in principle any succeeding constructive proof will generate a different version of the algorithm.

The representation of an algorithm synthesized with this method is a set of conditional equalities (equivalences for predicates) – we will refer to those as *clauses*. These clauses are in such a form that they can be very easily evaluated in a functional (or even imperative way) – as it can be seen in the examples later in the paper. Namely, if for instance a clause is of the form $\varphi \Rightarrow$

$F[\tau_1, \tau_2] = T$, where the variables a, b, A, B occur in φ, τ, T , then the condition φ is simple to evaluate over the domain of the elements only (for instance $a < b$), the term τ is such that it can be easily “inverted” by simple functions, and T contains only other easy to evaluate functions (like the constructors of lists) and sometimes additionally F applied to subterms of τ_1, τ_2 (typically the variables). By “inversion” we understand some functions which are able to easily extract the variables from the respective term, like e. g. if the term is $a \smile A$, then the functions are **Head** and **Tail**.

The proof based method and some experiments on algorithm synthesis on sets represented as monotone lists have been partially published in [4].

Theory Exploration.

We consider as true certain basic properties of set theory which are easy consequences of the Zermelo-Fraenkel axioms (a description of these axioms is given in [2]) and of the definitions of the basic predicates and functions on sets. We consider the well-known definitions for $\cup$ (union), $\cap$ (intersection), $\subseteq$ (inclusion), and $\setminus$ (difference) (see for instance [5]) and start to add properties in the

knowledge base which are necessary in the process of algorithm synthesis. In this way we explore the theory of sets (represented as monotone lists).

Conclusion

The synthesis is based on proving: we extract the algorithm as a set of conditional equalities (equivalences when synthesizing predicates) from the constructive proof of the isomorphism property of the respective function (predicate). The main contribution of this work is the identification of the inference rules, the proof strategies, and the necessary theory (based on the axioms of sets) which are necessary for the synthesis of the most important predicates and functions over sets: **Member**, **Inclusion**, **Union**, **Intersection**, and **Difference**. Moreover, we implement a specific prover in the *Theorema* system which is able to produce the proofs and also an algorithm extractor.

In comparison to [1] where the authors normalize sets and get sorted lists without duplications, in our work we use a bijective function for representation of sets. Moreover, their goal is to find induction principles in order to prove some propositions and our goal is to do synthesis by proving. We give the induction principles and discover algorithms.

One of the major advantages of our approach is that the algorithms for operating on sets represented as monotone lists are more efficient than the ones for operating on sets represented as non-sorted lists.

References

1. A. Bouhoula and F. Jacquemard. Automated Induction for Complex Data Structures. *CoRR*, abs/0811.4720, 2008.
2. B. Buchberger. Logic for Computer Science. RISC, Johannes Kepler University, Austria, Lecture notes, 1991.
3. 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.
4. I. Drămnesc and T. Jebelean. Automated Synthesis of Some Algorithms on Finite Sets. In *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Computer Society, September 2012. In Print.
5. Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming*, volume Volume 1: Deductive Reasoning. Addison-Wesley, 1985.
6. S. Wolfram. *The Mathematica Book*. Wolfram Media Inc., 2003.