A Case Study in Systematic Exploration of Tuple Theory

Isabela Drămnesc[1,⋆], Tudor Jebelean[2] and Adrian Crăciun[1,⋆⋆]

[1] West University of Timişoara,
Romania
[2] Research Institute for Symbolic Computation,
Johannes Kepler University,
Linz, Austria
`idramnesc@info.uvt.ro`,
`tjebelea@risc.uni-linz.ac.at`,
`acraciun@info.uvt.ro`

Abstract. We illustrate with concrete examples the systematic exploration of Tuple Theory in a bottom-up and top-down way.
In the bottom-up exploration we start from two axioms, add new notions and in this way we build the theory, check all the new notions introduced and prove some of them by the new prover which we created in the TH∃OREM∀ system.
In order to synthesize some algorithms on tuples (like e. g. insertion-sort) we use an approach based on proving. Namely, we start from the specification of the problem (input and output condition) and we construct an inductive proof of the fact that for each input there exists a solution which satisfies the output condition. The problem will be reduced to smaller problems, the method will be applied like in a "cascade" and finally the problem is so simple that the corresponding algorithm (function) already exists in the knowledge. The algorithm can be then extracted immediately from the proof. We present an experiment on synthesis of the insertion-sort algorithm on tuples, based on the proof existence of the solution. This experiment is paralleled with the construction (exploration) of the appropriate theory of tuples.
The main purpose of this research is to concretely construct examples of theories and to reveal the typical activities which occur in theory exploration, in the context of a specific application – in this case algorithm synthesis by proving.

1 Introduction

Mathematics is a technique of solving problems and the essence of mathematics is proving. By reasoning about the problems we obtain knowledge. This knowledge

⋆ Work partially supported by SciCom (Scientific Computing Platform) bilateral project, collaboration Romania-Austria.
⋆⋆ Work in the frame of the RISC Transnational Access Programme supported by the European Commission FP6 for Integrated Infrastructures Initiatives, project SCIEnce (Symbolic Computation Infrastructure for Europe), contract No. 026133.

can be managed for theory exploration and reasoning about the knowledge can be implemented on computers.

In [3], the author talks about automated theorem proving versus mathematical theory exploration using computers and explains that automated theorem proving does not mean only to prove a single theorem, it means that by making the proof we are doing the process of exploring theory using computers. And he explains all the steps that one should follow in the process of exploration using the TH∃OREM∀ system.

In our paper we present a complete exploration of Tuple Theory, using the TH∃OREM∀ system. Our Tuple Theory is similar with the one described in [22], [27], [3], [11]. In contrast with the existent Tuple Theory from the TH∃OREM∀ system, based on higher order predicate logic, in our approach we use first order predicate logic, because this increases the feasibility of proving.

We use the TH∃OREM∀ system [8] – see `www.theorema.org` (implemented on top of Mathematica [26]), because this is a software system for automated theorem proving which uses mathematical logic and offers support for the user to formalize and to introduce new mathematical notions, to introduce conjectures and to prove the new notions introduced, to extract algorithms from the proved theorems, and to use these algorithms for computing, solving, for writing articles for publications, writing courses, etc. The syntax is similar with mathematical logic and the proofs appear in a separate window and are generated in natural style, easy to read.

So, as a convention, in this paper all the definitions, axioms, propositions are written in the frame of the TH∃OREM∀ system.

1.1 Mathematical Theories

Mathematical theories (on computer or not) are built incrementally starting from a set of axioms, definitions and by adding propositions which we check and prove. A mathematical theory is expressed by the language, the knowledge base and the inference rules. The *language "$\mathcal{L}$"* contains a set of *predicate, function, and constant* symbols. The *knowledge base "$\mathcal{KB}$"* contains a set of predicate logic formulae over the language: *axioms, theorems, properties.* The *inference rules "$\mathcal{IR}$"* are rules that describe the reasoning system for the theory. The rules describe how to transform the current proof situation into a new one.

$$Theory = \langle \mathcal{L}, \mathcal{KB}, \mathcal{IR} \rangle$$

1.2 Systematic Exploration of Mathematical Theories

One can explore mathematical theories in a bottom-up or/and in a top-down way.

In the bottom-up exploration we start from a set of axioms and we add new notions that we check. We add properties describing the new notions introduced and we prove the propositions. In this way we build an entire theory. In section 2 we present the way we build the Tuple Theory in the bottom-up way.

In top-down exploration we start from the specification of the problem (input and output conditions) and we have to find an algorithm that satisfies the specification. This process is paralleled with exploring (building) the theory of tuples. In section 3 we describe, in general, the method that we use in order to synthesize algorithms and also an experiment on synthesis of the sorting algorithms for tuples, in particular the insertion-sort algorithm.

The process of systematic exploration of mathematical theories has a lot of advantages: all the information that the mathematician needs are visible and more accessible, he can use them without the need of thinking what is the notion that he has to know for proving something. Using computers it will be very useful for mathematicians because they do not have to do everything by hand and computers are much more faster than humans.

1.3 Related Work

Strategies for the systematic exploration of theories in a bottom-up and in a top-down way one can find in [3]. In fact many of the definitions and assumptions from Tuple Theory which we use in this paper are developed in that research work. Our previous work on bottom-up exploration of Tuple Theory is presented in [15].

Program synthesis in computational logic is a well studied field. For an overview of several approaches that have been investigated see [2]. A tutorial on synthesis of programs one can find in [20].

Deductive techniques for program synthesis are presented in [21] and techniques for constructing induction rules for deductive synthesis proofs are presented in [9].

Bruno Buchberger introduces a method for algorithm synthesis, called "lazy thinking", see [4] and also a model for theory exploration using schemes, see [5]. The "lazy thinking" method is implemented in the TH∃OREM∀ system [8], see [6]. In this approach, a "program schemata" (the structure of the solving term) is given to the synthesis procedure. In contrast, in our approach this schemata is also discovered by the procedure, only the inductive principle of the domain is given. Another approach for automatize the synthesis of logic programs was introduced in [18].

A case study on exploration of natural numbers based on schemes (a bottom-up exploration) is done in [16] and also the decomposition of natural numbers (top-down exploration) in [12].

In [13] one can see how to use program transformation techniques to develop six versions of sorting algorithms. For classifications of the sorting algorithms see [19], [17], [23].

The synthesis of sorting algorithms on tuples in higher order predicate logic, using the "lazy thinking" method you can find in [11]. A case study on synthesis of the Merge-Sort algorithm one can find in [24], [25].

In this paper we explore the Tuple Theory in a bottom-up and in a top-down way. In the top-down way we use a method for proving in the context of constructive synthesis that is applied like in a "cascade" to all the reduced problems. Other case studies using this method on synthesis of some algorithms for tuples you can find in [14].

In contrast with the other case studies on sorting algorithms existent in the literature, in particular sorting of tuples, in our approach we use a different method for synthesize the algorithms and the use as much as possible of first order logic. This method was introduced in 2008, for details see [10]. A case study of transforming recursive procedures into tail recursive using this method of synthesis one can find in [1].

2 Context

2.1 Tuple Theory - Bottom-Up Exploration

The Tuple Theory is expressed by the language, the knowledge base and the inference rules.

The *language "Tuple-$\mathcal{L}$"* is similar with the one described in [22],[7] and is a set of *predicate, function, and constant* symbols. The predicates are: the unary predicate "IT" that describes the type of the objects from the theory ("IT" stands for "Is Tuple") and the binary equality "=" predicate. The functions are: the binary function "$\smile$" that adds an element at the beginning of a tuple and the unary identity "Id" function. As a constant we consider the empty tuple represented by $\langle\rangle$.

$Tuple - \mathcal{L} = \langle\langle IT, =\rangle, \langle\smile, Id\rangle, \langle\langle\rangle\rangle\rangle$

The *Knowledge base* consists from a set of axioms: the generation and the unicity. These axioms are describing a tuple. The "generation" axiom states that the empty tuple is a tuple and that by adding an element to a tuple we also obtain a tuple. The "unicity" axiom states that by adding an element to a tuple we cannot obtain the empty tuple, and that two tuples $u \smile X$ and $v \smile Y$ are equal if and only if $u = v$ and $X = Y$. We will express them using the syntax from the TH∃OREM∀ system:

$$Axiom[\text{"}generation\text{"}, \begin{array}{c} (IT[\langle\rangle]) \\ \underset{a, IT[X]}{\forall} IT[a \smile X] \end{array}]$$

$$Axiom[\text{"}unicity\text{"}, \begin{array}{c} \underset{u, IT[X]}{\forall} (u \smile X \neq \langle\rangle) \\ \underset{u,v,IT[X],IT[Y]}{\forall} ((u \smile X = v \smile Y) \Rightarrow ((u = v) \wedge (X = Y))) \end{array}]$$

The induction axiom:

$$Axiom[\text{"}induction\ principle\text{"}, \\ (\mathfrak{F}[\langle\rangle] \bigwedge \underset{a, IT[X]}{\forall} (\mathfrak{F}[X] \Rightarrow \mathfrak{F}[a \smile X])) \Rightarrow \underset{IT[X]}{\forall} \mathfrak{F}[X]]$$

The *inference mechanism* is lifted from the induction axiom, where $\mathfrak{F}$ stands for any predicate.

The notations: $\langle\rangle$ is the empty tuple, $a \smile \langle\rangle$ is a tuple having one single element, $a \smile X$ is an element added to the tuple X. This notations allows to use only first order predicate logic.

Note that in our formalism we use square brackets as in $f[x]$ for function application and for predicate application, instead of the usual round parantheses as in $f(x)$.

For the bottom-up exploration of this Tuple Theory we start from the two axioms "generation" and "unicity" and introduce in the knowledge base new notions like: concatenation of 2 tuples, the function that replaces an element from a tuple, the definition for the minimum element from a tuple, the definition for adding an element at the end of a tuple, the definition for the reverse of a tuple, the definition of a sorted tuple (the merge-sort algorithm).

All these notions are checked with concrete examples in the TH∃OREM∀ system and some propositions are automatically proved by the new prover created in the system.

E.g. We check if the "generation" axiom that we introduce is correct:

$Compute[IT[2 \smile (4 \smile \langle\rangle)], using \to Axiom[\text{"generation"}]]$
$True$
$Compute[IT[today \smile (is \smile (Monday \smile \langle\rangle))], using \to Axiom[\text{"generation"}]]$
$True$

Introduce the definition for concatenation:

$$Definition[\text{"concatenation"}, any[u, X, Y], \left.\begin{array}{r}((\langle\rangle \asymp Y) = Y)\\ (Y \asymp \langle\rangle = Y)\\ ((u \smile X) \asymp Y = u \smile (X \asymp Y))\end{array}\right]$$

and test it:

$Compute[\langle\rangle \asymp (6 \smile \langle\rangle), using \to Definition[\text{"concatenation"}]]$
$6 \smile \langle\rangle$

$Compute[today \smile (is \smile \langle\rangle) \asymp a \smile (beautiful \smile (day \smile \langle\rangle)), using \to Definition[\text{"concatenation"}]]$
$today \smile (is \smile (a \smile (beautiful \smile (day \smile \langle\rangle))))$

Introduce the definition for the function that replaces an element from a tuple

$$Definition[\text{"replace"}, any[u, v, n, X], \left.\begin{array}{r}(replaceT[u \smile X, 0, v] = v \smile X)\\ (replaceT[u \smile X, n, v]) = (u \smile replaceT[X, n-1, v])\end{array}\right]$$

and test it using the Compute command:

$Compute[replaceT[(3 \smile (2 \smile \langle\rangle)), 0, 5], using \to Definition["replace"]]$
$5 \smile (2 \smile \langle\rangle)$

$Compute[replaceT[(3 \smile (3 \smile (3 \smile \langle\rangle))), 1, 5], using \to Definition["replace"]]$
$3 \smile (5 \smile (3 \smile \langle\rangle))$

$Compute[replaceT[2 \smile (2 \smile (2 \smile \langle\rangle)), 2, 6], using \to Definition["replace"]]$
$2 \smile (2 \smile (6 \smile \langle\rangle))$

$Compute[replaceT[today \smile (is \smile (First \smile (March \smile (2010 \smile \langle\rangle)))), 3, June],$
$using \to Definition["replace"]]$
$today \smile (is \smile (First \smile (June \smile (2010 \smile \langle\rangle))))$

Introduce the definition for the minimum element from a tuple:

$Definition["minimum", any[u, v, T],$

$$\left.\begin{array}{c} minelem[u \smile \langle\rangle] = u \\ (minelem[u \smile (v \smile T)] = minelem[u \smile T]) \Longleftarrow (u \leq v) \\ minelem[u \smile (v \smile T)] = minelem[v \smile T] \end{array}\right]$$

and test it with the Compute command:

$Compute[minelem[1 \smile (2 \smile \langle\rangle)], using \to \{Definition["minimum"]\}]$
1
$Compute[minelem[5 \smile (2 \smile \langle\rangle)], using \to \{Definition["minimum"]\}]$
2
$Compute[minelem[1 \smile (0 \smile (3 \smile \langle\rangle))], using \to Definition["minimum"]]$
0
$Compute[minelem[10 \smile (18 \smile (2 \smile (5 \smile \langle\rangle)))], using \to$
$Definition["minimum"]]$
2

Introduce the definition for adding an element at the end of a tuple:

$Definition["adding at the end", any[a, b, X],$

$$\left.\begin{array}{c} (\langle\rangle \frown b = b \smile \langle\rangle) \\ ((a \smile X) \frown b) = (a \smile (X \frown b)) \end{array}\right]$$

and test it with the Compute command:

$Compute[\langle\rangle \frown 6, using \to Definition["adding at the end"]]$
$6 \smile \langle\rangle$
$Compute[(2 \smile (3 \smile \langle\rangle)) \frown 6, using \to Definition["adding at the end"]]$
$2 \smile (3 \smile (6 \smile \langle\rangle))$
$Compute[(a \smile (b \smile (b \smile (c \smile \langle\rangle)))) \frown End, using \to$
$Definition["adding at the end"]]$
$a \smile (b \smile (b \smile (c \smile (End \smile \langle\rangle))))$

Introduce the definition for the reverse of a tuple:

$$\textit{Definition}[\textit{"reverse"}, \textit{any}[u, X], \left.\begin{array}{c} \textit{reverseT}[\langle\rangle] = \langle\rangle \\ \textit{reverseT}[u \smile X] = \textit{reverseT}[X] \frown u \end{array}\right]$$

and test it:

$$\textit{Compute}[\textit{reverseT}[1 \smile (8 \smile (3 \smile (4 \smile \langle\rangle)))], \textit{using} \to \{\textit{Definition}[\textit{"reverse"}], \textit{Definition}[\textit{"adding at the end"}]\}]$$
$$4 \smile (3 \smile (8 \smile (1 \smile \langle\rangle)))$$
$$\textit{Compute}[\textit{reverseT}[\textit{Hagenberg} \smile (\textit{of} \smile (\textit{Castle} \smile (\textit{the} \smile \langle\rangle)))], \textit{using} \to \{\textit{Definition}[\textit{"reverse"}], \textit{Definition}[\textit{"adding at the end"}]\}]$$
$$\textit{the} \smile (\textit{Castle} \smile (\textit{of} \smile (\textit{Hagenberg} \smile \langle\rangle)))$$

Introduce the merge-sort algorithm and all the notions that we need:

$$\textit{Definition}[\textit{"simple tuple"}, \textit{any}[u, X], \left.\begin{array}{c} \textit{is}[\langle\rangle] = \textit{True} \\ \textit{is}[u \smile \langle\rangle] = \textit{True} \\ \textit{is}[u \smile X] = \textit{False} \end{array}\right]$$

$$\textit{Definition}[\textit{"parts of the tuple"}, \textit{any}[u, v, X], \left.\begin{array}{c} \textit{ps}[\langle\rangle] = \langle\rangle \\ \textit{ps}[u \smile \langle\rangle] = u \smile \langle\rangle \\ \textit{ps}[u \smile (v \smile X)] = u \smile \textit{ps}[X] \\ \textit{pd}[\langle\rangle] = \langle\rangle \\ \textit{pd}[u \smile \langle\rangle] = \langle\rangle \\ \textit{pd}[u \smile (v \smile X)] = v \smile \textit{pd}[X] \end{array}\right]$$

$$\textit{Algorithm}[\textit{"combine the parts of the tuple"}, \textit{any}[u, v, X, Y], \left.\begin{array}{c} \textit{comp}[\langle\rangle, \langle\rangle] = \langle\rangle \\ \textit{comp}[\langle\rangle, v \smile Y] = v \smile Y \\ \textit{comp}[u \smile X, \langle\rangle] = u \smile X \\ (\textit{comp}[u \smile X, v \smile Y] = u \smile \textit{comp}[X, v \smile Y]) \Leftarrow u \leq v \\ \textit{comp}[u \smile X, v \smile Y] = v \smile \textit{comp}[u \smile X, Y] \end{array}\right]$$

Add the algorithm:

$$\textit{Algorithm}[\textit{"sorting merge-sort"}, \textit{any}[X], \left.\begin{array}{c} \textit{msort}[\langle\rangle] = \langle\rangle \\ (\textit{msort}[X] = X) \Leftarrow \textit{is}[X] \\ \textit{msort}[X] = \textit{comp}[\textit{msort}[\textit{ps}[X]], \textit{msort}[\textit{pd}[X]]] \end{array}\right]$$

and check if it is correct:

$$\textit{Compute}[\textit{msort}[7 \smile (4 \smile (5 \smile (2 \smile (6 \smile (1 \smile (3 \smile \langle\rangle)))))))], \textit{using} \to \{\textit{Definition}[\textit{"simple tuple(C)"}], \textit{Definition}[\textit{"parts of the tuple(C)"}], \textit{Algorithm}[\textit{"combine the parts of the tuple"}], \textit{Algorithm}[\textit{"sorting merge-sort"}]\}]$$
$$1 \smile (2 \smile (3 \smile (4 \smile (5 \smile (6 \smile (7 \smile \langle\rangle))))))$$

2.2 Reasoning—the new prover ”TuplesProverTM”

We created a prover, ”TuplesProverTM”, in the TH∃OREM∀ system that generates automatically some proofs of the propositions occurring in the process of theory exploration. This prover combines some of the existing provers with rewriting rules.

E.g. Introduce the Proposition ”simple tuple” and prove it using the theory ”tuples axioms” that contains the two axioms ”generation” and ”unicity” by the prover ”TuplesProverTM”:

$$Proposition["simple\ tuple", IT[a \smile (b \smile (c \smile (b \smile (b \smile \langle\rangle))))]]$$

$Prove[Proposition["simple\ tuple"], using \to Theory["tuples\ axioms"],$
$by \to TuplesProverTM]//Last//Timing$
$\{0.452 Second, proved\}$

Introduce the Proposition ”equality”:

$$Proposition\Big["equality", \underset{a, IT[X], IT[Y]}{\forall}((X = Y) \Rightarrow (a \smile X = a \smile Y))\Big]$$

and prove it by the prover ”TuplesProverTM”:

$Prove[Proposition["equality"], using \to Theory["tuples\ axioms"],$
$by \to TuplesProverTM]//Last//Timing$
$\{0.452 Second, proved\}$

Also, Introduce and prove the proposition:

$$Proposition\Big["singleton\ f", \underset{u, IT[X]}{\forall}(\langle u\rangle \asymp X = u \smile X)\Big]$$

$Prove[Proposition["singleton\ f"], using \to \{Theory["tuple\ 3"],$
$Definition["singleton"]\}, by \to TuplesProverTM, //Last$
$proved$

Details about the implementation and the proofs you can see in [15].

3 Synthesis of algorithms on tuples – Top-Down Exploration

In this section we give a general description of our approach and a case study in synthesis of the insertion-sort algorithm using the method that we present.

In Program synthesis we deal with the question ”Given a specification how one can find an executable program that satisfies the specification”. There are different methods for synthesize programs in computational logic, see [2]. We use the constructive synthesis. This means that from the specification we generate a conjecture, prove the conjecture and from the proof extract the algorithm.

The novel specific feature of our approach is applying a method for proving the conjecture like in a ”cascade”.

3.1 Proof based synthesis

We start from the specification of the problem (input and output condition). Given the *Problem Specification:*

Input: $I_F[X]$

Output: $O_F[X,Y]$

find the definition of F such that $\underset{X:I_F}{\forall} O_F[X,F[X]]$.

Please note that we use square brackets for function application and for predicate application as in $f[x]$, instead of the usual round parantheses as in $f(x)$. Moreover the quantifier variable may be qualified with a property, as in the formula above which stands for: $(\forall X)(I_F[X] \Longrightarrow O_F[X,F[X]])$.

We synthesize F by proving $\underset{X:I_F}{\forall}\underset{Y}{\exists} O_F[X,Y]$ by some induction principle (which is not automatically chosen).

We represent a tuple like an element added to a tuple. By the notations we understand: $\langle\rangle$ the empty tuple, by $a \smile \langle\rangle$ a tuple having one single element, by $a \smile X$ we understand an element added to the tuple X. This notation allows to use only first order predicate logic.

Base case: We prove $\underset{Y}{\exists} O_F[\langle\rangle,Y]$. If the proof succeeds to find a ground term R such that $O_F[\langle\rangle,R]$, then we know that $F[\langle\rangle]=R$.

Induction step: We take arbitrary but fixed a and X_0 (satisfying I_F), we assume $\underset{Y}{\exists} O_F[X_0,Y]$ and we prove $\underset{Y}{\exists} O_F[a \smile X_0,Y]$. We Skolemise the assumption by introducing a new constant Y_0 for the existential Y. If the proof succeeds to find a witness $T[a,X_0,Y_0]$ (term depending on a, X_0, and Y_0) such that $O_F[a \smile X_0, T[a,X_0,Y_0]]$, then we know that $F[a \smile X] = T[a,X,F[X]]$.

Extracting the algorithm: Finally the algorithm that we extract from the proof is:

$$F[X] = \begin{cases} R, & \text{if} X = \langle\rangle \\ T[a,X,F[X]], & \text{if} X \neq \langle\rangle \end{cases}$$

In the induction step the problem will be reduced into smaller and smaller problems in case we cannot find directly the witness. Then we create a new problem (that is simpler) and we apply the same method. This process is repeated and the method is applied like in a "cascade". Finally our problem is that simple that the appropriate functions for constructing the witness term can be found in the knowledge base. From the proof we extract the algorithm as a case distinction equality in which the witnesses found during the proof occur on the right hand side.

During the proof we detect various propositions that we need for the proof to succeed. We prove these propositions and after that introduce them into the knowledge and use them in the proof. We will not go into details in this paper for the process of inventing propositions. We just want to mention that by adding propositions to the knowledge base we explore the theory. So the process of proving is paralleled to the process of building the theory.

3.2 Case study: Synthesis of Insertion-Sort

Problem Specification:
$I_{\text{InsertionSort}}[X] : True$

$O_{\text{InsertionSort}}[X,Y] : \begin{cases} X \approx Y \\ \text{IsSorted}[Y] \end{cases}$

The notation $X \approx Y$ means that X has the same elements as Y and $IsSorted[Y]$ means that Y is the sorted version of X.

Our *knowledge base* contains the definitions: " $\approx$ " (having the same elements), "IsSorted", " $\triangleleft$ " (an element is member into a tuple), "dfo" (delete first occurrence of an element from a tuple):

$$\textit{Definition}\Big[\text{"} \approx \text{"}, \textit{any}[a,X,Y], \begin{array}{c} \langle\rangle \approx \langle\rangle \\ ((a \smile X) \approx Y) \Longleftrightarrow ((a \triangleleft Y) \wedge X \approx \textit{dfo}[a,Y]) \end{array}\Big]$$

$$\textit{Definition}\Big[\textit{"IsSorted"}, \textit{any}[a,b,X], \begin{array}{c} \textit{IsSorted}[\langle\rangle] \\ \textit{IsSorted}[a \smile \langle\rangle] \\ \textit{IsSorted}[a \smile (b \smile X)] \Longleftrightarrow ((a \leq b) \wedge \textit{IsSorted}[b \smile X]) \end{array}\Big]$$

$$\textit{Definition}\Big[\text{"} \triangleleft \text{"}, \textit{any}[a,b,X], \begin{array}{c} a \not\triangleleft \langle\rangle \\ a \triangleleft a \smile X \\ (a \neq b) \Longrightarrow ((a \triangleleft b \smile X) \Longleftrightarrow (a \triangleleft X)) \end{array}\Big]$$

$$\textit{Definition}\Big[\textit{"dfo"}, \textit{any}[a,b,X], \begin{array}{c} \textit{dfo}[a,\langle\rangle] = \langle\rangle \\ \textit{dfo}[a, a \smile X] = X \\ (a \neq b) \Longrightarrow (\textit{dfo}[a, b \smile X] = b \smile \textit{dfo}[a,X]) \end{array}\Big]$$

In order to synthesize the sorting algorithm we generate the conjecture (by convention in this paper this is written in the TH∃OREM∀ system, see [8]):

$$\textit{Proposition}\left[\textit{"problem InsertionSort"}, \underset{X\,Y}{\forall\exists} \begin{cases} X \approx Y \\ \textit{IsSorted}[Y] \end{cases}\right]$$

and try to prove it.

It is sufficient to prove the proposition "problem InsertionSort" because we do not take into consideration the type of the objects. We consider that all the objects are tuples and not checking the type does not influence the proof (that is why the input condition is $I_{\text{InsertionSort}}[X] : True$ and not $I_{\text{InsertionSort}}[X] : X is tuple$.

For proving the proposition "problem InsertionSort" we use the induction principle: $(P[\langle\rangle] \bigwedge \underset{a,X}{\forall}(P[X] \Longrightarrow P[a \smile X])) \Longrightarrow \underset{X}{\forall} P[X]$.

The induction principle is in second order logic, but by instantiation it becomes first order.

Base case: X=$\langle\rangle$

Prove $\underset{Y}{\exists}\begin{cases}\langle\rangle \approx Y\\ \text{IsSorted}[Y]\end{cases}$

Find witness $Y^?$ such that $\begin{cases}\langle\rangle \approx Y^? \ \text{(G1.1)}\\ \text{IsSorted}\left[Y^?\right] \ \text{(G1.2)}\end{cases}$

We take our first goal (G1.1) and try to do matching with the knowledge base. By matching $\langle\rangle \approx \langle\rangle$ (from the definition " $\approx$ ") with $\langle\rangle \approx Y^?$ we find *witness* $Y^?$ =$\langle\rangle$.

We take this witness and ckeck it in the second goal (G1.2).

Check for (G1.2): IsSorted[$\langle\rangle$] is true by definition "IsSorted".

So, we obtain a branch of the algorithm $\mathcal{S}[\langle\rangle]$=$\langle\rangle$. (*)

Induction Step:

Assume: $\begin{cases}X_0 \approx Y_0 & \text{(H1.1)}\\ \text{IsSorted}\,[Y_0] & \text{(H1.2)}\end{cases}$ and

Prove: $\begin{cases}a \smile X_0 \approx Y\\ \text{IsSorted}[Y]\end{cases}$ (G)

Find witness $Y^?$ such that: $\begin{cases}a \smile X_0 \approx Y^?\\ \text{IsSorted}\left[Y^?\right]\end{cases}$

We rewrite this by the definition " $\approx$ " and we must prove:

$$\begin{cases}a \triangleleft Y^? & \text{(G2.1)}\\ X_0 \approx \text{dfo}\left[a, Y^?\right] & \text{(G2.2)}\\ \text{IsSorted}\left[Y^?\right] & \text{(G2.3)}\end{cases}$$

Theory must contain the propositions:

$$\mathit{Proposition}\left[\text{"}\mathit{IsElem\ in\ Insertion}\text{"}, \underset{a,X}{\forall}(a \triangleleft \mathit{Insertion}[a, X])\right]$$

$$\mathit{Proposition}\Big[\text{"}\mathit{The\ same\ elements\ in\ Insertion}\text{"},$$
$$\underset{a,X,Y}{\forall}((X \approx Y) \Longrightarrow (X \approx \mathit{dfo}[a, \mathit{Insertion}[a, Y]]))\Big]$$

$$\mathit{Proposition}\Big[\text{"}\mathit{Sorting\ using\ Insertion}\text{"},$$
$$\underset{a,X}{\forall}(\mathit{IsSorted}[X] \Longrightarrow \mathit{IsSorted}[\mathit{Insertion}[a, X]])\Big]$$

We instantiate proposition "IsElem in Insertion" and we obtain:

a $\triangleleft$ Insertion[a, X_0] (H2.1).

By matching (H2.1) with (G2.1) we find *witness* $Y^?$=Insertion[a,Y_0].

Check for (G2.2): $X_0 \approx \text{dfo}\,[a, \text{Insertion}\,[a, Y_0]]$.

We instantiate "The same elements in Insertion" and we know:

$(X_0 \approx Y_0) \Longrightarrow (X_0 \approx$ dfo[a, Insertion[a, Y_0]]) (H2.2)

By (H2.2), (H1.1), by Modus Ponens we know:

$X_0 \approx$ dfo[a, Insertion[a, Y_0]] (H2.3) that is the same with (G2.2).

Check for (G2.3): IsSorted [Insertion $[a, Y_0]$]
Instantiate proposition "Sorting using Insertion" and we know:
IsSorted$[Y_0]$$\Longrightarrow$IsSorted[Insertion[a, Y_0]] (H2.4)
By (H2.4), (H1.2), by Modus Ponens we obtain:
IsSorted[Insertion[a, Y_0]] that is the same with (G2.3).

Case when we know the function "Insertion" The theory contains also the definition "Insertion":

$$\left.\begin{array}{c} \textit{Definition}[\textit{"Insertion"}, \textit{any}[a,b,X], \\ \textit{Insertion}[a, \langle\rangle] = a \smile \langle\rangle \\ (a \leq b) \Longrightarrow (\textit{Insertion}[a, b \smile X] = a \smile (b \smile X)) \\ (a > b) \Longrightarrow (\textit{Insertion}[a, b \smile X] = b \smile \textit{Insertion}[a, X]) \end{array}\right]$$

We match this definition and we obtain our witness and if the knowledge have this definition this means that it contains also the properties for satisfying the conditions to be the witness. So, we obtain: $\mathcal{S}$[a$\smile X_0$] = Insertion $[a, Y_0]$ (**)

By (*), (**) and by the transformation rules $X_0 \longrightarrow X$, $Y_0 \longrightarrow \mathcal{S}[X]$ we extract the algorithm: $\underset{a,X}{\forall}\begin{cases} \mathcal{S}[\langle\rangle] = \langle\rangle \\ \mathcal{S}[a \smile X] = \text{Insertion}[a, \mathcal{S}[X]] \end{cases}$

Case when we do not know the function "Insertion" In this case the problem is reduced into a simpler problem (synthesize the function insertion).

We write X as an element a added to the tuple T. The input condition is extended (by adding that T is sorted) and the output condition remains the same.

Reduced Problem:

Problem Specification: (X: a, T)
$I_{\text{Insertion}}[a, T]$: IsSorted[T]
$O_{\text{Insertion}}[a, T, Y] : \begin{cases} a \smile T \approx Y \\ \text{IsSorted}[Y] \end{cases}$
Generate the conjecture:

$$\left.\begin{array}{l} \textit{Proposition}[\textit{"reduced problem Insertion"}, \\ \underset{a,T}{\forall}(\textit{IsSorted}[T] \Longrightarrow \underset{Y}{\exists}((a \smile T \approx Y) \wedge \textit{IsSorted}[Y])) \end{array}\right]$$

Prove the proposition "reduced problem Insertion" by induction over T.
We use the Induction Principle Head/Tail (Decomposition)
T: $\begin{cases} \langle\rangle & (P[\langle\rangle]) \\ b \smile T & (P[T] \Longrightarrow P[b \smile T]) \end{cases}$
We take a arbitrary, but fixed and we start to prove.
Base case: T=$\langle\rangle$

Prove $\underset{Y}{\exists}\begin{cases} a \smile \langle\rangle \approx Y \\ \text{IsSorted}[Y] \end{cases}$

Find witness $Y^?$ such that $\begin{cases} a \smile \langle\rangle \approx Y^? & \text{(G3)} \\ \text{IsSorted}\left[Y^?\right] & \text{(G4)} \end{cases}$

We take our first goal (G3) and try to do matching with the knowledge base.

Theory must contain the proposition:

$$\textit{Proposition}\left[\textit{"reflexivity in} \approx \textit{"}, \underset{X}{\forall}(X \approx X)\right]$$

We use the proposition "reflexivity in $\approx$ " and by matching with (G3) we find witness $Y^? = a \smile \langle\rangle$.

Check for (G4): IsSorted[a$\smile$ $\langle\rangle$]. This is true by the definition "IsSorted".

So, we obtain Insertion[a, $\langle\rangle$] = a $\smile$ $\langle\rangle$. (*I*)

Induction Step: T: b$\smile$T

Assume: IsSorted[T]$\Longrightarrow \underset{Y}{\exists}\begin{cases} a \smile T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (H*4)

Prove: IsSorted[b$\smile$T]$\Longrightarrow \underset{Y}{\exists}\begin{pmatrix} a \smile (b \smile T) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G*4)

For proving (G*4) we assume IsSorted[b$\smile$T] (H4) and

prove: $\underset{Y}{\exists}\begin{pmatrix} a \smile (b \smile T) \approx Y \\ \text{IsSorted}[Y] \end{pmatrix}$ (G4)

By (H4) and "property of sort", by Modus Ponens we obtain IsSorted[T] (H5). The theory must contain the proposition:

$$\textit{Proposition}\left[\textit{"property of sort"}, \underset{a,X}{\forall}(\textit{IsSorted}[a \smile X] \Longrightarrow \textit{IsSorted}[X])\right]$$

From (H5), (H*4), by Modus Ponens we know: $\underset{Y}{\exists}\begin{cases} a \smile T \approx Y \\ \text{IsSorted}[Y] \end{cases}$ (H6)

In (H6), by Skolem we obtain: $\begin{cases} a \smile T \approx Y_0 & \text{(H6.1)} \\ \text{IsSorted}\,[Y_0] & \text{(H6.2)} \end{cases}$

By the definition " $\approx$ " we rewrite (H6.1) into: $\begin{cases} a \triangleleft Y_0 & \text{(H7.1)} \\ T \approx \text{dfo}\,[a, Y_0] & \text{(H7.2)} \\ \text{IsSorted}\,[Y_0] & \text{(H6.2)} \end{cases}$

For proving (G4):

Find witness $Y^?$ such that $\begin{cases} a \smile (b \smile T) \approx Y^? \text{(G4.1)} \\ \text{IsSorted}\left[Y^?\right] \text{(G4.2)} \end{cases}$

Alternative I:

Using proposition "reflexivity in $\approx$" we do matching in (G4.1) and obtain witness: $Y^? = a \smile (b \smile T)$.

Check in (G4.2): IsSorted[a$\smile$(b$\smile$T)].

By definition "IsSorted" we have to prove: (a$\leq$b) $\wedge$ IsSorted[b$\smile$T].

IsSorted[b$\smile$T] is true by (H4).

So, (a$\leq$b)$\Longrightarrow$ $Y^? = a \smile (b \smile T)$ and we obtain:

Insertion[a,b$\smile$T]=a$\smile$(b$\smile$T) iff a$\leq$b. (*II*)

Alternative II:

By IsSorted$\,[Y_0]$ (H6.2), a $\triangleleft$ Y_0 (H7.1) and by $\neg$(a$\leq$b) (from the proposition "order"), and if we consider that a is the smallest element from Y_0, we can use the property "Sorting II" and we obtain that IsSorted$\,[b \smile Y_0]$ (H5).

The theory must contain the propositions:

$$\mathit{Proposition}\left[\mathit{"order"}, \underset{a,b}{\forall}\begin{matrix}(\neg(a \leq b)) \Longrightarrow (a > b)\\ a > b \Longrightarrow a \neq b\end{matrix}\right]$$

$$\mathit{Proposition}[\mathit{"Sorting\ II"},$$
$$\left.\underset{a,b,X}{\forall}((\mathit{IsSorted}[b \smile X] \wedge (b \lhd (b \smile X)) \wedge (a \leq b)) \Longrightarrow \mathit{IsSorted}[a \smile (b \smile X)])\right]$$

We do matching of (H5) with (G4.2) and we obtain witness $Y^{?} = b \smile Y_0$.

Check in (G4.1): a$\smile$(b$\smile$T)$\approx b \smile Y_0$.

By the definition " $\approx$ " we rewrite this into:

$$\begin{cases} a \lhd b \smile Y_0 \qquad (G4.12)\\ b \smile T \approx \mathrm{dfo}\,[a, b \smile Y_0] \quad (G4.13)\end{cases}$$

By the proposition "order" we know that a$\neq$b (H5.1).

By (H5.1) and by the definition " $\lhd$ " in (G4.12) we must prove that
a $\lhd$ Y_0 that is true by (H7.1).

Check for (G4.13):

By the definition "dfo" and by (H5.1) we rewrite (G4.13) into:

$b \smile T \approx b \smile \mathrm{dfo}\,[a, Y_0]$ (G4.14)

By proposition "dfo $\approx$ " we have to prove:
$T \approx \mathrm{dfo}\,[a, Y_0]$ that is true by (H7.2).
Theory must contain the proposition:

$$\mathit{Proposition}\left[\mathit{"dfo} \approx \mathit{"}, \underset{a,T,X}{\forall}((a \smile T \approx a \smile X) \Longrightarrow (T \approx X))\right]$$

So, our witness is correct: $Y^{?} = b \smile Y_0$.

We obtain Insertion[a,b$\smile$T]=b$\smile Y_0$ iff a>b. (*III*)

From (*I*), (*II*), (*III*) and by the transformation rules
$Y_0$$\longrightarrow$Insertion[a, T], T$\longrightarrow$X we obtain the algorithm:

$$\underset{a,b,X}{\forall}\left(\begin{matrix}\mathit{Insertion}[a, \langle\rangle] = a \smile \langle\rangle\\ \mathit{Insertion}[a, b \smile X] = \begin{cases} a \smile (b \smile X), a \leq b\\ b \smile \mathit{Insertion}[a, X]\ , a > b\end{cases}\end{matrix}\right)$$

4 Conclusion and Future Work

We presented in this paper the systematic exploration of Tuple Theory. In section 2 we describe the way how one can build the theory of tuples starting from two axioms, the "generation" and the "unicity" axioms, by adding new notions and properties of these notions in the frame of the TH∃OREM∀ system. We check all the new notions introduce and also we prove some of the propositions introduced by the prover created in the TH∃OREM∀ system. In section 3 we describe how one can build the theory of tuples by synthesizing algorithms on tuples, in particular the synthesis of insertion-sort algorithm. The problem was reduced into simpler problems and we apply the same method like in a "cascade".

During the proof we introduce in the knowledge base the propositions that we need to continue the proof, in this way the process of proving is paralleled to the process of theory exploration.

The method for synthesize algorithms presented here and also the case study on synthesis of the insertion-sort algorithm are subject for implementation in the frame of the TH∃OREM∀ system. We plan to extend this work by investigating other tuple operations and other sorting algorithms (like e.g. merge-sort algorithm).

References

1. P. Audebaud and L. Chiarabini. New Development in Extracting Tail Recursive Programs from Proofs. In *Pre-Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'09)*, Coimbra, Portugal, September 9-11, 2009, 2009.
2. D. Basin, Y. Deville, P. Flener, A. Hamfelt, J. F. Nilsson, and Mathematical Modelling. Synthesis of Programs in Computational Logic. In *Program Development in Computational Logic*, pages 30–65. Springer, 2004.
3. B. Buchberger. Theory Exploration with Theorema. *Analele Universitatii Din Timisoara, Ser. Matematica-Informatica*, XXXVIII:9–32, 2000.
4. B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. *Analele Universitatii din Timisoara, Seria Matematica - Informatica*, XLI:41–70, 2003. special issue on Computer Science - Proceedings of SYNASC'03.
5. B. Buchberger. Algorithm Supported Mathematical Theory Exploration: A Personal View and Strategy. In J. Campbell B. Buchberger, editor, *Proceedings of AISC 2004*, volume 3249 of Springer LNAI, pages pages 236–250, 2004.
6. B. Buchberger and A. Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. *Electr. Notes Theor. Comput. Sci.*, 93:24–59, 2004.
7. B. Buchberger and A. Craciun. Algorithm synthesis by lazy thinking: Using problem schemes. In *SYNASC 2004*, pages 90–106. Mirton, Timisoara, Romania, 2004.
8. 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.
9. A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing Induction Rules for Deductive Synthesis Proofs. *Electron. Notes Theor. Comput. Sci.*, 153(1):3–21, 2006.
10. L. Chiarabini. Extraction of Efficient Programs from Proofs: The Case of Structural Induction over Natural Numbers. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Local Proceedings of the Fourth Conference on Computability in Europe: Logic and Theory of Algorithms (CiE'08)*, pages 64–76, Athens, Greece, June 15-20, 2008, 2008.
11. A. Craciun and B. Buchberger. Algorithm Synthesis Case Studies: Sorting of Tuples by Lazy Thinking. Technical Report 04-16, RISC–Linz, Austria, October 2004.
12. A. Craciun and M. Hodorog. Decompositions of Natural Numbers: From A Case Study in Mathematical Theory Exploration. In D. Petcu, D. Zaharie, V. Negru, and T. Jebelean, editors, *SYNASC07*, 2007.

13. J. Darlington. A Synthesis of Several Sorting Algorithms. *Acta Inf.*, 11:1–30, 1978.
14. I. Dramnesc and T. Jebelean. Proof Based Synthesis of Sorting Algorithms. Technical Report 10-17, RISC Report Series, University of Linz, Austria, 2010.
15. I. Dramnesc, T. Jebelean, and A. Craciun. Case Studies in Systematic Exploration of Tuple Theory. Technical Report 10-09, RISC Report Series, University of Linz, Austria, May 2010.
16. M. Hodorog and A. Craciun. A Case Study in Systematic Theory Exploration: Natural Numbers. Technical Report 07-18, RISC–Linz, Austria, October 2007.
17. 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.
18. I. Kraan, D. Basin, and A. Bundy. Middle-Out Reasoning for Logic Program Synthesis. In *Proc. 10th Intern. Conference on Logic Programing (ICLP '93)* (Budapest, Hungary), pages 441–455, Cambridge, MA, 1993. MIT Press. Also available as Technical Report MPI-I-93-214.
19. K. K. Lau. A Note on Synthesis and Classification of Sorting Algorithms. *Acta Informatica*, 27:73–80, 1989.
20. K. K. Lau and G. Wiggins. A Tutorial on Synthesis of Logic Programs from Specifications. In P. Van Hentenryck, editor, *Proc. 11th Int. Conf. on Logic Programming*, pages 11–14. MIT Press, 1994.
21. Z. Manna and R. Waldinger. Synthesis: Dreams ? programs. *IEEE Transactions on Software Engineering*, 5:294–328, 1979.
22. Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming*, volume Volume 1: Deductive Reasoning. Addison-Wesley, 1985.
23. Susan M. Merritt. An Inverted Taxonomy of Sorting Algorithms. *Commun. ACM*, 28(1):96–99, 1985.
24. D. R. Smith. A Problem Reduction Approach to Program Synthesis. In *IJCAI*, pages 32–36, 1983.
25. D. R. Smith. Top-down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27(1):43–96, 1985. (Reprinted in "Readings in Artificial Intelligence and Software Engineering", Eds. C. Rich and R. Waters, Morgan Kaufmann Pub. Co., Los Altos, CA, 1986, pp.35-61).
26. S.Wolfram. *The Mathematica Book*. Wolfram Media Inc., 2003.
27. R. Waldinger Z. Manna. *The Logical Basis for Computer Programming*, volume Volume 2: Deductive Systems. Addison-Wesley, 1990.