Systematic Exploration of Tuples Using Schemes

Codresi Ana-Cristina *
ccodresi@ieat.ro

24th July 2006

Abstract

In this paper we report a case study of computer supported exploration of the theory of tuples, using a theory exploration model based on knowledge schemes, proposed by Bruno Buchberger.
We illustrate with examples from the exploration:

- the invention of new concepts (functions, relations) in the theory, using knowledge schemes,
- the invention of new propositions, using proposition schemes.

1 Introduction

The systematic exploration of mathematical theories refers to developing a theory by adding new concepts to it. The exploration consists from the following steps:

1. Invention of mathematical notions (i.e definitions for functions) using recursive knowledge schemes.
2. Invention and verification of propositions.
3. Invention of problems, and solving the problems.

We consider the exploration of tuples. We choose this case study because tuples and Computer Science are close related, tuples are used in almost all branches of Computer Science. The case study is carried out in the THEOREMA system, and we will use its notational convention.

In section 2 we present the context of our case study: the logic, the language of tuples, our notion of a theory, describe the schemes used in the exploration process.

Sections 3 and 4 contain various examples from the exploration: adding notions, conjectures, problems using schemes, and lifting knowledge to the inference level.

*Work supported by the European Commission's Marie Curie Actions Programme: European Reintegration Grants, Project No.MERG-CT-2004-012718

2 Context

2.1 Language, Knowledge Base, Inference Rules, Theories

To express mathematical theory, we use a first order predicate logic language with equality. The first order language $\mathcal{L}$ describing the theory, is a triple: $\mathcal{L} = \langle \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$, where $\mathcal{P}$ is the set of predicate symbols, $\mathcal{F}$ is the set of function symbols, $\mathcal{C}$ is the set of constants of the theory.

The *knowledge base* $\mathcal{KB}$ of the theory consists of a collection of first-order formulae, built over the language (axioms, properties).

The *inference mechanism* $\mathcal{IR}$ of the theory consists of the reasoning engines corresponding to the theory. This will include for any theory, the first order predicate calculus, rewriting and specific rules.

In conclusion, in our context, a *theory* $\mathcal{T}$ is a triple $\mathcal{T} = \langle \mathcal{L}, \mathcal{KB}, \mathcal{IR} \rangle$.

2.2 The Theory of Tuples

We will describe the theory of tuples as we had introduced it.
The language of tuples,

$$\mathcal{L}[0] := \bullet language[\bullet constants[\bullet[\langle\rangle]], \bullet functions[\bullet[Identity], \bullet[\smile]], \bullet predicates[\bullet[=], \bullet[is-tuple], \bullet[atom]]]$$

Under the intended models for the theory certain domain elements are atoms and certain elements are tuples. The atom relation atom(x) is true if x denotes an atom, and false otherwise. The tuple relation tuple(x) is true precisely if x denotes a tuple. Also the value of the insertion function u$\smile$ x is the tuple obtained by inserting the atom u at the beginning of the tuple x.
The knowledge base, $\mathcal{KB}$, corresponding to the initial theory consists from the equality axioms, tuples axioms and the induction principle, formulated in the context of the tuples theory:

Axioms["equality", any[*is-tuple*[x, y, z]],
$x = x$ "reflexivity"
$(x = y) \Leftrightarrow (y = x)$ "symmetry"
$((x = y) \wedge (y = z)) \Rightarrow (x = z)$ "transitivity"
$((x = y) \wedge p[x]) \Leftrightarrow p[y]$ "pred subst for p"
$(x = y) \Rightarrow (f[x] = f[y])$ "funct subst for f"]

Axioms["tuples:generation", any[*is-tuple*[x], *atom*[u]],
is-tuple[$\langle\rangle$] "generation:empty"
is-tuple[$x \smile u$] "generation:insertion"]

Axioms["uniqueness:tuples", any[*is-tuple*[x], *is-tuple*[y], *atom*[u], *atom*[v]],
$u \smile x \neq \langle\rangle$ "uniqueness:empty"
$u \smile x = v \smile y \Rightarrow (u = v) \wedge (x = y)$ "uniqueness:insertion"]

Axiom["induction principle",

$$((F[\langle\rangle] \wedge (\underset{atom[u],is\text{-}tuple[x]}{\forall}(F[x] \Rightarrow F[u \frown x])) \Rightarrow \underset{is\text{-}tuple[x]}{\forall}(F[x]))].$$

Remark. Axiom ["equality"], "pred subst for p", "funct subst for f", and Axiom["induction principle"] are *axiom schemes*, and these cannot be expressed in first order logic. In each case we indicated that certain symbols are metavariables, where the argument is a free variable. In conclusion, we cannot use these axioms as them are, so, we will lift them to the inference level, as below.

The inference mechanism, $\mathcal{IR}$, corresponding to the theory, consists from the structural induction rule, general predicate logic inference rules, and inference rules for equality (simplification, rewriting).

The induction rule can be lifted to the inference level, as it follows:

$$\frac{KB \vdash Fx \leftarrow \langle\rangle \qquad is\text{-}tuple[y]abf, atom[u]abf, Kb \cup Fx \leftarrow y \vdash Fx \leftarrow u \frown y}{KB \vdash \underset{is\text{-}tuple[x]}{\forall} F},$$

that is, in order to prove the universally quantified goal, prove the base case (substituting $\langle\rangle$ for the variable), then assume the goal formula true for an arbitrary but fixed value, and prove the goal for $u \frown y$.
Anallogus, the axioms schemes for equality are also lifted to the level of inference rules.

2.3 Knowledge Schemes

Knowledge schemes are formulae that capture mathematical knowledge at various levels of abstraction. These schemes are stored in libraries of schemes, and are used by instantiating them with symbols from the language of the theory. We also work at constructing a global library of schemes, at the highest level of abstraction (not depending on the theory), and other libraries that are dependant of the theory.

To formulate knowledge schemes we need higher order function and predicate variables. The THEOREMA language allows higher order formulae, so schemes can be formulated. The schemes are formulae used for inventing definitions, proposition, problems and algorithms.

Even thow the schemes are higher order formulae, when we instantiate them with symbols from our theory we return to first order logic.

We will give some examples of knowledge schemes used for inventing mathematical notions in the theory of tuples.

The first examples, are definitions knowledge schemes that generated the concatenation function symbol, the member relation, and the reverse function. All of the schemes that we used are recursive schemes, and some of them generated binary function and relation symbols, others generated unary function symbols.

$$schUnaryFunction := \bullet[(\underset{f,h}{\forall}(is\text{-}simple\text{-}rec\text{-}unary\text{-}function[f, g, \langle\rangle]) :\Leftrightarrow$$
$$(\underset{is\text{-}tuple[x],atom[u]}{\forall}((f[\langle\rangle] = \langle\rangle) \wedge (f[u \frown x] = h[f[x], u])))))]$$

$$schBinaryFunction := \bullet[\underset{f,g,h}{\forall}(is\text{-}simple\text{-}rec\text{-}binary\text{-}function[f, g, h]) :\Leftrightarrow$$
$$(\underset{is\text{-}tuple[x,y],atom[u]}{\forall}((f[\langle\rangle, y] = g[y]) \wedge (f[u \frown x, y] = h[u, f[x, y]])))]$$

$$schBinaryRelation := \bullet[(\underset{R,C}{\forall}(is\text{-}simple\text{-}rec\text{-}binary\text{-}relation[R]) :\Leftrightarrow$$
$$(\underset{is\text{-}tuple[x],atom[a,b]}{\forall}((R[a,\langle\rangle]) \wedge (R[a, b \frown x] \Leftrightarrow C[R[a,b], R[a,x]])))]$$

The second examples are some algebraic schemes, used to introduce new structural propositions with respect to the concatenation function symbol.

$$schSemigroup := \bullet[\underset{p,bin\text{-}op}{\forall}(is\text{-}semigroup[p, bin\text{-}op] \Leftrightarrow) \underset{p[x,y,z]}{\forall}$$
$$p[bin\text{-}op[x, y]]$$
$$bin\text{-}op[x, bin\text{-}op[y, z]] = bin\text{-}op[bin\text{-}op[x, y], z]]$$

3 Introducing the Theory of Tuples: Exploration Round 0

Tuples, L[0]: Axioms
*Remark.*The notations L[0], L[1],...,L[n], stand for the language at the n-th exploration step. Still this notation must and will be improved.
At this moment our theory contains the axioms for equality, and the axioms for tuples, including the induction principle. In Theorema we will collect all the notions under the *Theory* header.
Tuples, L[o]: Exploration
At this level of exploration our theory includes only the axioms for equality, and the axioms for tuples (the generation axioms and the uniqueness axioms)
As an interaction between the predicate symbols, atom and is-tuple, and the function symbol $\frown$, we obtain an important property, the decomposition property.
Decomposition:

$$\mathsf{Proposition}[\text{"tuples:decomposition"}, \mathsf{any}[is\text{-}tuple[x]],$$
$$x \neq \langle\rangle \Rightarrow \underset{atom[v],is\text{-}tuple[x]}{\exists}(x = v \frown y)]$$

4 Tuples: Exploration Round 1. New Function Symbol

4.1 Current Knowledge

The language used in the first step of exploration is the same with the initial language presented earlier, because no new mathematical notions were introduced.

4.2 Analysis of construction

Analizing the tuples construction we see that we can introduce two new functions, head[x] and tail[x], where head[x] denotes the first atom of a nonempty tuple x, and tail[x] denotes the tuple of all but the first atom of a nonempty tuple x.
The functions tail and head are defined by:

- Definition[head]

$$\begin{array}{l}\mathsf{Definition}[\text{"}head\text{"}, \mathsf{any}[is\text{-}tuple[x], atom[u]],\\ head[u \smile x] = u \qquad\qquad \text{"def of head"}]\end{array}$$

- Definition[tail]

$$\begin{array}{l}\mathsf{Definition}[\text{"}tail\text{"}, \mathsf{any}[is\text{-}tuple[x], atom[u]],\\ tail[u \smile x] = x \qquad\qquad \text{"def of tail"}]\end{array}$$

The interactions between these two functions with the predicates atom and is-tuple are given by the following properties:

- The Sort of head

$$\begin{array}{c}\mathsf{Proposition}[\text{"sort of head"}, \mathsf{any}[is\text{-}tuple[x]],\\ (x \neq \langle\rangle) \Rightarrow (atom[head[x]])]\end{array}$$

- The Sort of tail

$$\begin{array}{c}\mathsf{Proposition}[\text{"sort of tail"}, \mathsf{any}[is\text{-}tuple[x]],\\ (x \neq \langle\rangle) \Rightarrow (is\text{-}tuple[tail[x]])]\end{array}$$

4.3 Exploration: Adding a New Function Symbol

At this moment we have explored all the notions of our theory, and all the possible interactions between them. We will like to see what we will obtain using recursive schemes, for functions and for relations.

4.3.1 Current knowledge

Once new notions are introduced, the language used in the previous step of exploration is extended to a new language, of the following form:

$$\begin{array}{c}\mathcal{L}[1] \leftarrow \mathcal{L}[0] \cup \{head, tail\} :=\\ \bullet language\,[\bullet constants[\bullet\,[\langle\rangle]], \bullet functions[\bullet\,[Identity]\,, \bullet\,[\smile]\,, \bullet\,[head]\,, \bullet\,[tail]],\\ \bullet predicates[\bullet\,[=]\,, \bullet\,[is\text{-}tuple]\,, \bullet\,[atom]]\end{array}$$

4.3.2 Using Recursive Definitions/Algorithms Schemes

In this section we will give some examples of how the schemes are applied in the steps of exploration.

We begin the exploration with the scheme for binary recursive function symbols presented in the previous section. For this we will use a special function called *UseScheme*, which takes as arguments the general knowledge scheme and the list of all possible combinations between all the notions already introduced in the theory and generates all the possible functions that can be constructed with these symbols.

$$\begin{array}{c}\mathsf{possibleSubsts} := \{\{f \rightarrow \bullet\,[f1]\,, g \rightarrow \bullet\,[Identity]\,, h \rightarrow \bullet\,[Insertion]\}\,,\\ \{f \rightarrow \bullet\,[f2]\,, g \rightarrow \bullet\,[Head]\,, h \rightarrow \bullet\,[Insertion]\}\,,\\ \{f \rightarrow \bullet\,[f3]\,, g \rightarrow \bullet\,[Tail]\,, h \rightarrow \bullet\,[Insertion]\}\end{array}$$

$$\mathsf{newConcepts} := \mathsf{Map}\left[\mathsf{UseScheme}\left[schBinaryFunction, \sharp\right], \mathsf{possibleSubsts}\right]$$

For good reason we will choose the function f1, and explore it. We will rename it *Concatenation*.f2 is a symbol function that does not exist because of the sort matching, and f2 is a function symbol that inserts the atoms of the first tuple to the tail of the second tuple. We must make the observation that after executing these instructions the variable *newConcepts* contains a list of all the possible function symbols that can be generated. From this list we extract only those function symbols that we need, and in this case we only extract the concatenation function symbol.

4.3.3 Exploration of the New Function Symbol Introduced

We have seen how we can automatically obtain the Concatenation function, using a binarry recursive scheme. We will introduce the Concatenation, as it follows:

$$\mathsf{Definition}\left["concatenation", \mathsf{any}\left[is\text{-}tuple\left[x\right], is\text{-}tuple\left[y\right], atom\left[u\right]\right],\right.$$
$$\langle\rangle \asymp y = y$$
$$\left.(u \smile x) \asymp y = u \smile (x \asymp y)\right]$$

The exploration continues by verifying if the set of all tuples with the *Concatenation* is some algebraic structure. And we see that the tuples with *Concatenation* verify:

- the associativity property

$$\mathsf{Proposition}["\text{concatenation:associativity}", \mathsf{any}[is\text{-}tuple[x], is\text{-}tuple[y], is\text{-}tuple[z]],$$
$$(x \asymp y) \asymp z = x \asymp (y \asymp z)]$$

- there exists a neutral element, and this is $\langle\rangle$
- about the existence of the inverse... we can prove that:

$$\mathsf{Proposition}["\text{concatenation:annihilation}", \mathsf{any}[is\text{-}tuple[x], is\text{-}tuple[y]],$$
$$(x \asymp y = \langle\rangle) \Rightarrow ((x = \langle\rangle) \wedge (y = \langle\rangle))]$$

So, not for all tuples exists the inverse (related to *Concatenation*), actually anly the empty tuple has an inverse.
In general the tuples concatenation is not commutative, so our conclusion is that the tuples with the *Concatenation* verify the *monoid* algebraic structure. There are some other properties, which are interactions between the *Concatenation* and the other existing symbols.

In the same way we obtain the *Member Relation*, and the *Reverse Function* symbol.

$$\mathsf{Definition}["memRel", \mathsf{any}[atom[u], atom[v], is\text{-}tuple[x]],$$
$$\neg(u \in \langle\rangle)$$
$$(u \in (v \smile x)) \Leftrightarrow ((u = v) \vee (u \in x))]$$

$$\mathsf{Definition}["revFunct", \mathsf{any}[is\text{-}tuple[x], atom[u]],$$
$$reverse[\langle\rangle] = \langle\rangle$$
$$reverse[u \smile x] = reverse[x] \asymp \langle u\rangle]$$

5 Implementation

In this section we present the new provers that we have implemented. The new prover that we have written in THEOREMA implements the induction over tuples. It is called TTIP and it is being used in three user provers. The names of this user provers are:

1. *NewTupleIndProver*, the prover combining natural deduction, rewriting and induction over tuples. It is used to prove general formulae.
2. *NewTupleEqIndProver*, the prover combining rewriting and induction over tuples. It is used to prove equations.
3. *NewTuplePredIndProver*, the prover combining natural deduction and induction over tuples.

These three provers are used in proving the propositions from the tuples theory, depending on each situation. We are still working at the implementation of these provers.

6 Conclusions

In this paper we give several examples of introducing mathematical notions in the theory of tuples. These examples represent only a part of the theory that we want to define. Still, further exploration must be done in order to complete the developement of the tuples theory.

First of all, we propose to prove the complete induction principle. The complete induction principle is important because any sentence we can prove by complete induction principle we can also prove by induction principle, but the proof made by the induction principle may be more complex. Therefore after proving the complete induction principle, we can use it to prove all the propositions from the theory.

Secondly, we want to study the interactions between the nonnegative integers and the tuples, and to see what new functioon and relation symbols produces. For introducing new symbols in the theory, new knowledge scheme must be written, expanding in this way the existing knowledge schemes base.

Finally, we want to improve and develop the induction provers for the tuples theory. Some induction provers have already been implemented having as the inference rule the traditional induction principle and using the natural deduction rules for induction principle and rewriting rules for mathematical formulae. Such provers are: *NewTupleIndProver, NewTupleEqIndProver, NewTuplePredIndProver*. For developing new induction provers we propose to use the complete induction principle as an inference mechanism, achieving in this way shorter proofs. Also, we want to develop sorting functions for tuples.

References

[1] J.R. Schoenfield, *Mathematical Logic*, Addison-Wesley Publishing Co, SUA, 1967

[2] Z. Manna and R. Waldinger, *The Deductive Foundations of Computer Programming*, Addison-Wesley Publishing Co, SUA, 1993

[3] B. Buchberger, *Algorithm- supported mathematical theory exploration: A personal view and strategy*, Lecture Notes in Artificial Intelligence, Springer, 7th Conference on Artificial Intelligence and Symbolic Computation (Research Institute for Symbolic Computation, Hagenberg, Austria) (Proceedings of AISC 2004, 16 September).