Exploring and Combining Deductive and Inductive Reasoning in Large Libraries of Formalized Mathematics *

Josef Urban

Dept. of Theoretical Computer Science and Mathematical Logic
Charles University
Malostranské nám. 25, Praha, Czech Republic

A dissertation submitted for the degree of Doctor of Philosophy

August 5, 2004

*The work described in this thesis was partially supported by the CALCULEMUS European research training network (HPRN-CT-2000-00102)

Abstract

This thesis is about making large libraries of formalized mathematics available to modern automated theorems provers (ATPs) [Robinson and Voronkov 2001], experimenting with ATPs and evaluating them on such large libraries, and combining ATPs with machine learning methods to make them practically useful for theorem proving in such large libraries.

We argue here that the scientific fields involved in this work, i.e. the field of computer-checked formalization of mathematics, the field of automated deduction (deductive reasoning), and the field of machine learning (inductive reasoning) can profit from this combination and that such a combination can be significant for the fields of mathematics and artificial intelligence in general.

The main step towards such combination taken in this thesis is the development of the first experimental version of the MPTP [Urban 2004] system (Mizar Problems for Theorem Proving). MPTP translates the world's largest library of formalized mathematics - the Mizar Mathematical Library (MML) [Rudnicki 1992] - to a format suitable for modern first-order ATP systems. It also implements a number of functions over the translated library, which allow generating of a very large number of ATP problems in different ways. Particularly, a first version of an efficient signature-filtering method derived from the analysis of the Mizar verifier is used to remove redundant context formulas for the main experiments with reproving the Mizar theorems by ATP systems. These experiments reveal that about one third of all Mizar theorems can be proved by an ATP system, if we supply the premises used in the MML proof.

Because of the large number of theorems and definitions in the MML, the complete signature-filtering based techniques are generally insufficient, if the premises for a proof of some formula are not explicitely specified. We suggest and implement a machine learning solution to the problem of selecting suitable premises from such a large repository. The SNoW [Carlson et all 1999] multi-target machine learning system is trained to associate selected features of MML formulas with the premises which are most likely to be useful in their proofs. We experimentally evaluate the precision of the trained system, and show that in combination with the signature-filtering techniques and the SPASS [Weidenbach 2001] ATP system, it is possible to prove fully automatically about one seventh of all Mizar theorems.

Acknowledgments

First of all, I am greatly indebted to Prof. RNDr. Petr Štěpánek, DrSc, for supervising and supporting my PhD work, for both giving me the freedom to experiment with my own ideas, and for pushing me in the right direction when it was necessary. Supervising me and my work was not always an easy task, and I really appreciate his patience and wisdom that helped me during the time of my PhD studies.

Though it is unfortunately too late for prof. Zdeněk Renc to read this, I would also like to express here my thanks to him. He was the opponent of my master thesis, and since then showed continuous interest and support for my work.

A very large part of my work is based on the Mizar system started by prof. Andrzej Trybulec, and developed under his supervision at the University of Bialystok, Poland. Many thanks to him and also his colleagues and students for the numerous discussions of the Mizar system and formalization of mathematics in general. Thanks also for inviting me to Bialystok and their care, hospitality and friendship during my stay there.

My work uses many ideas from the ILF project, lead by prof. Ingo Dahn. He and his colleagues were very kind and helpful in providing me with the ILF system and its installation. A very important person both in the Mizar and in the ILF context is dr. Czeslaw Bylinski, on whose work for the ILF system I could build with the MPTP system.

Prof. Piotr Rudnicki and Geoff Sutcliffe have discussed with me many ideas presented in this work and shown their interest and support. This also holds true about my fellow PhD students Petr Pudlák and Jiří Vyskočil.

Thanks also to Nick Rizzollo and prof. Dan Roth for their help and suggestions concerning my work with the SNoW Learning Architecture.

Finally, thanks to the organizers of the CALCULEMUS European research training network (HPRN-CT-2000-00102) for making my visits to the University of Bialystok possible, and to the Czech METACentrum project for providing me with the computing resources which were necessary for my experiments.

Introduction

In the field of artificial intelligence and particularly in its *artificial reasoning* part, two large subfields can be observed today. The first one is the field of *deductive reasoning*, represented by automated or interactive theorem proving, proof checking, constraint solving, etc. The other one is the field of *inductive reasoning*, represented by machine learning, data mining, knowledge discovery in databases, etc.

There is a fundamental difference between the paradigms of these two fields. Deductive reasoning is mainly concerned with inference algorithms which are correct, and possibly also complete, in the mathematical framework that they use. This holds e.g. for the DPLL procedure [Davis and Putnam 1960] used in the propositional SAT solvers, or for the various refinements of the resolution and paramodulation based methods and tableaux based methods used in the automated theorem provers for first order logic. Logical correctness is also the main issue in the interactive theorem provers and proof checkers. In short, the deductive reasoning is centered around the notion of a formal symbolic proof given in some logical framework.

The notion of a formal proof is usually missing in the contemporary inductive reasoning methods. Here we are typically trying to induce a plausible hypothesis from some previous experience, evaluate such hypothesis statistically against some test data, and use it afterwards e.g. for classification or decision support. The complete semantics of the domains to which inductive techniques are applied do not have to be (and usually are not) specified, in some typical domains like e.g. natural language processing this would be very hard. On one hand, this means that the current inductive methods are much more robust than the deductive ones, i.e. one incorrect data item has only marginal effect on the inductive reasoning process, which is far from true in deductive reasoning. On the other hand, this causes that the inductive systems usually lack the capability to ultimately show that their hypotheses are in some sense correct, or that they are related to each other in some logical sense, which seems to be an important part of human reasoning.

We therefore believe that for the development of smarter artificial intelligence systems, it is necessary to provide databases (or rather knowledge bases), which would be large enough for the inductive reasoning methods, and also enough semantically rich, so that deductive methods could be applied. The most natural choice for such knowledge bases seem to be libraries of mathematics, where the very ideas of deduction and completely specified

semantics originated, and number of deductive tools are available.

Unfortunately, most of the currently available mathematics uses a lot of natural language[1] and its detailed semantics is therefore usually not available for machine processing. The field of *formalization of mathematics* tries to change this state, by introducing formal computer-understandable mathematical languages that are used to formalize parts of mathematics and verify them by computers. In this way, large libraries of formal mathematics are being created.

In the effort to be as human-understandable and user-friendly as possible, such large libraries however usually use much richer mathematical formalism than that used by the most efficient tools for automated deduction[2]. Thus the quest for the cooperation between the inductive and the deductive AI methods leads to another quest, which is no less interesting and potentially rewarding. This is the quest for the cooperation between formalization of mathematics and efficient automated theorem proving[3].

At this moment we could go further and identify a number of other interesting goals originating from or related to the two tasks mentioned above. Some of them are discussed later in this thesis. However the main goal of this thesis is to provide initial answers to these two problems. Particularly, we describe and implement the first experimental version of the MPTP system (Mizar Problems for Theorem Proving), which translates the world's largest library of formalized mathematics - the Mizar Mathematical Library (MML) - to a format suitable for modern first-order automated theorem provers. The MPTP translation is then used as a knowledge base for a combined inductively-deductive system, which uses the previous proof experience extracted from the knowledge base to guide the automated theorem proving of arbitrary formulas expressed in the MPTP language.

Despite the fact that the two systems are at their first versions, and their implementation tries to be as straightforward as possible, their first results

[1]The term *mathematical vernacular* is often used for the language of mathematical books and articles.

[2]Note that while there are a number of very efficient SAT solvers, constraint solvers or first-order automated theorem provers using clausal logic, there are practically no good automated provers working with full first-order formulas today, to say nothing about automated provers for higher-order logic.

[3]And also vice versa, we believe that for successful fully automated theorem proving in large mathematical libraries, the combination with inductive reasoning techniques is necessary.

obtained are encouraging. About one third of the proofs of theorems in the MML can be constructed by an ATP system, if the user provides the high-level advice in the form of other MML theorems and definitions. About one seventh of all MML theorems can be proved fully automatically by the combined inductively-deductive architecture trained on the previous MML proofs, without any user advice. This is a task which cannot be handled by any current purely deductive or purely inductive system, and the result suggests that new tasks like searching for new proofs of available theorems, or discovery of advanced new theorems can be realistically tried with such combined systems.

We hope that the work and the results presented in this thesis will provide an important feedback to the scientists working on automated deduction systems, formalization of mathematics or inductive systems. The first message is that their systems are already strong enough and that the libraries are large enough for the both kinds of cooperation to be realistically attempted. The second message is that it really makes sense (and will be appreciated by the end-users of the combined systems), if more attention is paid to those features of their systems which facilitate such cooperation. We also hope that the results presented here will stimulate other artificial intelligence researchers to contribute new and better combined architectures to this exciting scientific field, and possibly try to push its borders beyond the scope of pure mathematics.

Structure of this thesis

This thesis consists of two articles written by Josef Urban. The first one, "MPTP - Motivation, Implementation, First Experiments" [Urban 2004] has been accepted to the 2004 special issue of the Journal of Automated Reasoning on first-order theorem proving. It contains the description of the first version of the MPTP system (section 2), first results of the experiments with reproving the Mizar theorems by the SPASS ATP system (section 4.1), the implementation of the Mizar Proof Advisor based on the SNoW learning system (section 4.2), the measurements of the standard machine-learning performance of this system, and first results of the combined inductively-deductive architecture consisting of the Mizar Proof Advisor, MPTP signature filtering and the SPASS prover.

The second article, "Translating Mizar for First Order Theorem Provers", is the original longer version of the paper [Urban 2003] published in 2003 in the Proceedings of MKM 2003, Lecture Notes in Computer Science, Vol. 2594. This article describes in detail the translation of the Mizar language to untyped first-order format suitable for current automated theorem provers. It also contains as an appendix the complete grammar of the Mizar language, which can be used as a reference during reading the examples of the Mizar texts in this thesis.

We hope that the selected order of articles will make reading this thesis more easy, than if we started with a formal description of the Mizar language, that went on with the details of the translation, and only after that described the achieved results. Though the work described here really had to start with the translation of Mizar to the formats suitable for first-order ATP systems, and it has also taken most of the time spent on this work, the first (JAR) article should provide enough high-level information for many people who are not interested in the detailed description of Mizar. The second article and the Mizar syntax can than be used as a detailed reference for the first article. For readers who prefer the bottom-up approach, it might be helpful to read first some of the introductions to Mizar mentioned in the second article and then continue with the second and the first article.

The appendices added at the end of this thesis try to complete the articles with more detailed examples and information. We show the SQL structure of the MPTP result database and the manual page of the MPTP top-level problem generating script, describing many of its functionalities. On a selected Mizar problem we give an example of the transformation that we perform,

showing the original Mizar proof, the generated reproval task with complete background information and the reproval task after the signature filtering, the hints given by the Mizar Proof Advisor for proving the problem, and the corresponding theorem proving problem. The SPASS proofs are shown for all these inputs. Note that the complete MPTP system as well as the Mizar distribution on which it is based is included on the CD, which is included in this thesis. It also includes the complete result files of the experiments described here and the technical descriptions and scripts telling how exactly the experiments were conducted. We hope that this will enable the readers to look up the results or re-run the experiments in which they are particularly interested. All the software included in the MPTP system, except the Mizar-based binary program fo_tool, is covered by the GNU Public License. The translated MML and the fo_tool program should be distributed under the same terms as the Mizar distribution.

References

[Carlson et all 1999] Carlson A. J., Cumby C. M., Rosen J. L. and Roth D. [1999], SNoW User's Guide. UIUC Tech report UIUC-DCS-R-99-210.

[Davis and Putnam 1960] Martin Davis , Hilary Putnam, A Computing Procedure for Quantification Theory, Journal of the ACM (JACM), v.7 n.3, p.201-215, July 1960

[Robinson and Voronkov 2001] A. Robinson and A. Voronkov, eds., Handbook of Automated Reasoning, The MIT Press, September 1, 2001

[Rudnicki 1992] Rudnicki P. [1992], An Overview of the Mizar Project, Proceedings of the 1992 Workshop on Types for Proofs and Programs, Chalmers University of Technology, Bastad.

[Urban 2003] Urban J. [2003], Translating Mizar for First Order Theorem Provers. In Andrea Asperti, Bruno Buchberger, James Davenport (eds.), Mathematical Knowledge Management, Proceedings of MKM 2003, LNCS 2594.

[Urban 2004] Josef Urban. MPTP - Motivation, Implementation, First Experiments. Accepted to Journal of Automated Reasoning, First-Order Theorem Proving Special Issue. editors Ingo Dahn, Deepak Kapur and Laurent Vigneron. Kluwer Academic Publishers (supposed publication: end of 2004).

Available online at http://kti.ms.mff.cuni.cz/~urban/MPTP/mptp-jar.ps.gz.

[Weidenbach 2001] Weidenbach C. [2001], SPASS: Combining Superposition, Sorts and Splitting, in A. Robinson and A. Voronkov (Eds.), Handbook of Automated Reasoning, Vol II, Chapter 27, pp. 1965–2013, Elsevier Science and MIT Press.

Table of Contents

MPTP - Motivation, Implementation, First Experiments

Josef Urban
Dept. of Theoretical Computer Science
Charles University
Malostranske nam. 25, Praha, Czech Republic
*(*urban@kti.ms.mff.cuni.cz*)*

Abstract. We describe a number of new possibilities for current theorem provers, that arise with the existence of large integral bodies of formalized mathematics. Then we proceed to describe the implementation of the MPTP system, which makes the largest existing corpus of formalized mathematics available to theorem provers.

MPTP (Mizar Problems for Theorem Proving) is a system for translating the Mizar Mathematical Library (MML) into untyped first order format suitable for automated theorem provers, and for generating theorem proving problems corresponding to MML. The first version generates about 30000 problems from complete proofs of Mizar theorems, and about 630000 problems from the simple (one-step) justifications done by the Mizar checker. We describe the design and structure of the system, the main problems encountered in this kind of system, their solutions, current limitations, and planned future extensions.

We present results of first experiments with reproving the MPTP problems with theorem provers. We also describe first implementation of the Mizar Proof Advisor (MPA) used for selecting suitable axioms from the large library for an arbitrary problem, and again, present first results of this combined MPA/ATP architecture on MPTP.

Keywords: MPTP, Mizar, ATP, MPA

1. Motivation

1.1. Theorem Provers, Assistants, and Mathematical Libraries

The situation in the fields of theorem provers, proof assistants and formalization projects is at the moment roughly following:

There are Otter-style automated theorem provers (ATPs) actively being developed, usually working in untyped first order predicate calculus and usable in automatic mode for all kinds of problems that are "simple enough". These provers are being constantly improved, both by devising new theoretical approaches (e.g. the superposition calculus recently, various decision procedures for various fragments of the logic, etc.), and by more "practical" implementation techniques, like special purpose indexing techniques. Obviously, an important factor in the growing usability of ATP systems is also Moore's Law. The proof of the Robbins conjecture found by EQP (McCune 1997) can be used as

an evidence that current ATP systems are already capable of solving much more than just simple toy problems.

Proof assistants (Wiedijk 2003) are used to help the creation of computer-checked proofs, and usually include wider functionality, ranging from proof presentation solutions, library browsing and searching tools, to the actual proof checkers or tactical provers. The underlying logic used in such systems is usually more complicated than just simple untyped predicate calculus, and often uses type systems to provide early error checking and some notion of type "obviousness" (people do not want to prove all the time that natural number is also real number). The tactical provers used there usually implement smaller proof steps, that can often be used as building blocks of more complex user-programmed tactics. Sometimes even full-strength automated theorem proving tactics are thus available, but they are usually very simple and inefficient in comparison with ATP systems, and sometimes sacrifice the more complicated aspects of the logic (e.g. the type system) to get a simpler implementation.

Some proof assistants are mainly used and designed for the task of the formalization of pure mathematics, building large libraries (similar in certain aspects to large software libraries) of theorems and definitions, reusable in more and more advanced theories. The largest of such libraries is the Mizar Mathematical Library (MML) built with the Mizar system (Rudnicki 1992). It is also the most "purely mathematical" library when assessing the contents, and unlike some other libraries, its foundations (Tarski-Grothendieck set theory) are very close to ZFC, used as a foundation for most of the current mainstream mathematics.

The main objective of such formalization efforts is usually the formalization itself (e.g. providing computer-checked proof of the fundamental theorem of algebra, Birkhoff's variety theorem or Jordan curve theorem), but this also serves the long-term dream of formalizers, that with "battle-tested", fine-tuned and user-friendly systems, and big libraries, the advantages of computer mathematics (proof checking and assistance, reliable semantic searching, etc.) will eventually prevail over the current mainstream brain-to-TEXway of authoring mathematics.

1.2. Using ATP Systems on Mathematical Libraries

So far, there has been very little cross-fertilization between ATP systems and large formalization projects. The reasons of this state are not clear to the author. One can be the fact that really large formalized libraries have only appeared in about last ten years, or may be that the two scientific groups do not much pay attention to each other,

ATP people being the "strong AI" to whom the formalization task may seem easy and theoretically uninteresting, while formalization people confronted with the complexity of this "easy" task may regard ATP as rather theoretical and toy systems, used for toy problems, and not "real" mathematics.

There have been attempts to use ATP systems with software libraries (Schumann 2001), but still the improvements to ATP systems are usually of a very general and universally applicable nature (e.g. ordered resolution improves the general resolution in a very general way). Such improvements are very good, however there might be other, more specific methods, how to improve ATP systems in various mathematical domains. Recently, some ATP systems started to explore these possibilities, the most advanced example is probably the E prover (Schulz 2002; Schulz 2001), which uses machine learning methods on several levels, to optimize its behavior on various classes of problems.

However, at this point the problem of datasets usable for training of such systems appears. The standard TPTP (Sutcliffe and Suttner 1998) library is good for measuring the improvement in the general methods used by ATP systems, and it can probably even be used to learn problem classifications, in notions like number of input clauses, their average weight, number of symbols, etc., to conjecture best proving strategies on such problem classes. But it probably does not make too much sense to try learning of domain specific optimizations on TPTP, e.g. considering whether the problem symbols are typical set-theoretical symbols or typical algebraic symbols. This can be the reason, why the current learning methods usually abstract from the symbol level, paying attention to the abstracted term structure at best.

The situation is very different with large structured formalized ensembles like MML, where the symbols and relations among them are very stable and organized, and play decisive role for consistency and usability. So machine learning techniques going to the level of symbols (e.g. learning the best symbol and term orderings in various domains) make much more sense here, while methods using more abstract representations of terms or clauses obviously can be tried too, with the additional possibility of using them to find new similarities between different theories.

Another opportunity coming with large structured libraries, is exactly their structure. In MML, very advanced theories are really developed from the ground (i.e. Tarski-Grothendieck axioms). There are no other axioms allowed in MML, all constructions are really carried out. This means e.g. that consistent construction of integers, rational and real numbers is done before any calculus takes place, then going into more advanced fields like Lebesgue measure theory in a book-style

presentation of the definitions and theorems. There are currently almost 800 articles in MML, building on each other. There are many intertwining lines of development, one of the most cited being the project of formalization of the book "Compendium of Continuous Lattices" (Bancerek 2000), where about 60 percent of the book has already been formalized.

This structure can be used for another kind of optimization of ATP systems. We have the possibility to follow the proofs of theorems, expanding the lemmas and references used in them to arbitrary level with their own proofs, thus creating hierarchy of increasingly difficult problems, where lemma conjecturing (perhaps mostly expressed as splitting in current provers) is the key to success. The lemmas introduced in MML proofs can then be used as a vast repository of examples usable for improving this capability of theorem provers by machine learning methods, maybe even suggesting some new domain-independent approaches.

Similar opportunity comes with the rich structure of MML definitions. There are nearly 8000 definitions introduced in MML. Humans introduce definitions to simplify the problem they are solving, effectively hiding some part of it in the definition. This is a method that probably has not even been used so far in ATP systems[1] which are just at the point of exploring the rules for unfolding the definitions that are already present. Adding such methods to ATP systems would just itself be a very significant step in their development, taking them from purely deductive tools to a more combined inductive/deductive architecture, closing the gap between them and more inductive systems like e.g. AM (Lenat 1979; Lenat 1982), and providing a new approximation to the ideal of a"universal AI system".

A big challenge is the type system used in the libraries. There are fast type-inferencing and type-checking mechanisms implemented in proof checkers. It is a nontrivial problem, to include similar mechanisms as parts of complete strategies used in ATP systems, however, as some first experiments show, it may be rewarding.

More generally, most proof assistants (even the non-programmable ones, like Mizar) have some level of automation. Apart from the type-inferencing, this may e.g. include some efficient decision or evaluation procedures, e.g. simple arithmetical evaluation in Mizar. It is also a big challenge to try to deal with such problems efficiently, within the frame of the complete methods used by theorem provers.

[1] Actually, some advanced skolemization techniques used in FLOTTER (Nonnengart and Weidenbach 2001) already introduce new definitions to simplify problem representations.

However, the probably most pressing new task and field of research, which can only appear when a rich and consistent set of notions has been developed, and many facts proved about them, is the problem of choosing premises. This happens when a user of the library comes and asks the simple question "Is assertion X valid?". It would probably be very difficult for current provers to use e.g. all theorems from the library for indiscriminate proof search for X or its negation, since it is in the nature of resolution proving, that redundant premises make the task harder. So the problem is to find the smallest relevant set of premises available in the library, while keeping the chance of success (i.e. completeness) sufficiently high. Again, statistical and machine learning methods using previous experience from the library are likely to be used for such tasks, however also adding more structure to provers' clause databases could complement this, e.g. with strategies like "add the next most promising external premise, when the prover runs for too long".

Finally, let us finish this enumeration of some new possibilities with a note on ATP-based theorem discovery over such libraries. Current refutational provers work by saturating the given theory, with the hope of finding contradiction, caused by the negated conjecture added to the premises. During the search process, subsumption is usually used to keep only the strongest versions of clauses generated by the saturation. However, the prover can also be used to saturate some initial theory to certain level, i.e. without any explicit negated conjecture. Inspecting the kept (i.e. unsubsumed) clauses after some time of such saturation run, checking them for subsumption with the whole library (e.g. with systems like MoMM (MoMMUrl; Urban 2004)), and possibly filtering them with some other criteria (e.g. weight), might be used for adding new useful clauses to the library. It would be very interesting to see how good (e.g. in comparison with the human-designed library) are the facts derived in this way, and again, to try some optimizations, e.g. with respect to the initial set of theorems, prover settings, etc., or even try to combine this with the possibility of introducing definitions, mentioned above, again attempting more general "theory exploring" AI systems.

So much for the ideas and arguments for closer cooperation between ATP systems and large formalization projects. In the next part of the article we describe the first version of the MPTP system, which is designed exactly with the aim to enable such cooperation between ATP systems and MML.

2. Mizar Problems for Theorem Proving

MPTP is available online at
`http://alioth.uwb.edu.pl/twiki/bin/view/Mizar/MpTP`.
The main packed distribution has about 70 MB and unpacks to about 100MB. It is possible to download only the basic distribution (about 300 kB) without libraries, and build the main (possibly customized) libraries from the Mizar system.

2.1. Overview of MPTP

MPTP 0.1 at the time of writing this description consists of the following parts:

- the main Mizar-to-ATP translation tool (fo_tool)
- Makefiles and some very simple scripts creating the translated library from fo_tool's output
- the translated library, accessible both as Prolog files and as Berkeley DB files
- Perl scripts accessing the library as Berkeley DB files, generating proof problems and providing other important functionality, like signature filtering or results parsing
- the generated proof problems

Additionally, an SQL (MySQL) database of results with Web interface is used for collecting and analysis of prover results. This is not included in the system distribution.

2.2. Mizar-to-ATP translation tool

The main Mizar-to-ATP translation tool (fo_tool) is a standalone program, based on the Mizar implementation (written in objective Pascal). Since the Pascal sources of the Mizar system are only available to members of the Association of the Mizar Users, we only distribute a Linux binary, executable on x86 architectures. This limitation is only important for those who want to build the translated library for themselves from the Mizar distribution. MPTP is distributed with the library already built, so fo_tool is not necessary for its normal use.

The tool is similar to the Mizar "exporter" program, which is used to put the exportable parts of Mizar articles (theorems, definitions,

etc.) into the internal Mizar database[2]. fo_tool takes a Mizar article as input, and produces several files containing the translated information about various Mizar constructors, theorems, definitions and clusters exported from the article. This functionality corresponds closely to the Mizar exporter.

Additionally, fo_tool also collects information about complete proofs of exported Mizar theorems, which covers some statistics about the proof (its length expressed as the length of the list of all references[3] (including local references and repeated occurrences) as they appeared in the proof), and the set of all external references used in the proof. The set of external references later serves as the smallest set of premises, from which the theorem should be provable. However, it has to be almost always enlarged by adding some implicit context (background) information (e.g. type rules), that the Mizar checker uses for checking the inferences.

Even though we are taking only the smallest set of formulas necessary for proving the task, these tasks can be quite difficult for ATP systems, since proofs of Mizar theorems are usually quite long, and additionally, provers are not capable of using the necessary background information (e.g. type rules) as efficiently as the Mizar checker. So to have a more simple group of problems, fo_tool also exports all Mizar "Simple Justification" problems, i.e. the simplest Mizar inference steps that usually look like:

```
then a*a <= a*b & a*b < b*b by A1,AXIOMS:25,REAL_1:70;
```

which tells Mizar that the fact "`a*a <= a*b & a*b < b*b`" should be directly provable (keyword "*by*") from the previous formula (keyword "*then*"), which here was "`0 < b & a <= b`", from the private reference A1, which here was "`0 <= a & a < b`" and from theorem 25 in article AXIOMS and theorem 70 in article REAL_1. The Mizar checker proceeds by negating the conjecture, and employing number of methods[4], to try to derive a contradiction from the negated conjecture and the referenced premises.

[2] While MML denotes the collection of all Mizar articles, created by humans, the internal Mizar database is another part of the system, used for efficient storage and fast access to those parts of Mizar articles, that can be reused in other articles.

[3] Inference steps in Mizar are usually justified by giving labels of other formulas, from which the new inferred fact should follow. These formulas are either inferred locally earlier in the proof (private references), or taken from MML (library references).

[4] The probably most detailed available description of the methods used by the Mizar checker is in (Wiedijk 2000), some specific methods are also discussed in (Naumowicz and Byliński 2002).

The Mizar checker is not a complete theorem prover, since speed is also important for proof assistants[5], and additionally, it is not desirable from the point of the legibility of the proofs (important e.g. when generalizing some theory, or for some future educational applications (Dahn 2001)), to have the checker too strong. So the exported checker (Simple Justification) problems, should generally be quite easy for ATP systems, though exceptions to this might occur, again due to efficient handling of the background information in the checker, or due to its quite strong congruence-closure based equality handling.

We do not go here into the details of the translation of various Mizar constructs, and of the more complicated logical framework, into untyped first order format suitable for ATP systems. This is explained on examples from MML in (Urban 2003). However, it should be at least noted here, that direct translation into the DFG format (Hähnle et all 1996) used by the SPASS prover (Weidenbach 2001) was chosen for the first MPTP version. We realize, that the TPTP format (Sutcliffe and Suttner 1998) is probably most widely used today, and in fact, its support is already built into fo_tool, because it shares some code with the MoMM project, which uses the TPTP format. But it seems that the SPASS prover performs best on the translated problems[6], probably because of its autodetection of sort theories and use of semantic blocking (Ganzinger et all 1997) of ill-typed inferences, which can often efficiently approximate the fast Mizar handling of its large type and cluster background theories. As mentioned above, designing the translation and making it work is a nontrivial task, and we need the best prover available for testing, and we want to minimize the number of translation layers, at least in the beginning. The dfg2tptp tool (available in SPASS distribution) can be used now to translate DFG tasks to TPTP format, but it is possible that we will make TPTP (or rather the newly suggested TSTP (Sutcliffe et all 2003)) the default format in the future, or will support more than one output format.

2.3. The Exported Library

Most current ATP formats (including the commonly used subset of DFG syntax) are Prolog-based. This is quite advantageous, because

[5] Complete MML has now about 60 MB, and its complete processing takes less than 1 hour on 2GHz Pentium 4, which is quite important for doing large-scale revisions of MML.

[6] In the initial experiments we compared SPASS 2.0 with E 0.7 using a very low (4 seconds) timelimit. and SPASS solved 1000 problems more (8727 against 7737). Geoff Sutcliffe has recently tried the Vampire prover with 300s timelimit and proved 12828 problems, however this is hard to compare with the SPASS results given in the Results section, since different hardware was used.

problem inputs or databases can be quickly analyzed by loading them into one's favourite Prolog system. One of the goals in the design of the translated library, is to maintain this possibility.

However, we also have to think about fast and memory efficient access to various parts of the library, since the number of creatable problems is very large (about 30000 for theorem problems and about 630000 for checker problems), and for problem creation, we want to be able to implement efficiently some advanced functions, like signature filtering. Such functions require indexing, which together with the need for memory efficiency, call for a database (e.g. SQL) approach to the library. Such an approach was also previously used for handling translated MML in the ILF system (Dahn and Wernhard 1997), from which MPTP takes much inspiration.

The problem with such approach is that tables in database systems are usually stored in some internal binary format, definitely not Prolog parsable. This can be solved by various means, and the approach we have chosen is again motivated by the effort to have the system as simple and transparent as possible.

The library is now a collection of several files, usually containing formulas in DFG format, expressing some part of the translated MML structure. So all translated theorems are in one file, all definitions in another, etc., and these files are Prolog readable (though sometimes quite big). We keep small index file (also in Prolog format), telling for each library file F, and each Mizar article A, at which point of F the translated items from A are placed. Since most Mizar items (e.g. theorems, definitions, constructors, etc.) are already numbered by the Mizar system (e.g. REAL_1:70 is 70th theorem in article REAL_1), and the naming scheme used by our translation respects this numbering (again, the naming scheme is dealt with in more detail in (Urban 2003)), it is thus usually very simple and fast (constant time) to compute a position of some item in a library file.

This approach now takes care of most of the indexing problems, necessary for fast access into the library files. The memory efficiency is solved by accessing the library files as simple Berkeley DB databases of the RECNO (record number) type. Berkeley DB is capable of working directly with the normal text format for this kind of databases, so there is no need for any other internal binary versions of the library files. This has the additional advantage, that Berkeley DB is today a very standard part of most Unix-like systems or distributions, used by many applications, so users do not have to go through additional installation process, which would be the case for SQL systems or Prolog implementations.

The creation of the library is automated by using a large Makefile, which is parametrized by a list of Mizar articles that should be processed. This is usually just the list of all Mizar articles in their MML processing order, however using just some initial segment of this list (e.g. first 100 articles) is possible, for creating smaller versions of the translated library. It takes about 2 hours on Pentium 4 2GHz, to create the complete libraries from a Mizar distribution, most of the time being taken by fo_tool.

Additionally, the library contains input files for checker problems, again in a Prolog format. Because of the number of checker problems, these files can be quite large (several MB) even for a single article, and would occupy about 1 GB, if not compressed. So for space efficiency, we keep them compressed in a special directory. Decompression and cleanup of these files are handled by the problem generating scripts.

2.4. Problem Generating Scripts

Problems creation and other MPTP functions are implemented in about 5000 lines of documented Perl modules and scripts, that make use of the standard DB_File Perl module for interfacing Berkeley DB files. The architecture tries to be simple and extensible.

The basic MPTP utilities (Perl module MPTPUtils.pm) provide database access to the translated library, functions for creating the basic background theory for articles, based on their environment directives, and functions for problem printing.

As the results of first experiments confirm, a very important part of the system is the default signature filtering module (Perl module MPTPSgnFilter.pm), based on reasonings about the Mizar checker. Signature filtering functions get the explicit premises for some problem, and the basic background theory created for the problem's article (i.e. formulas encoding type, cluster, and other information available in the article), and starting only with the explicit premises, they try to cut off all unnecessary background formulas. This is done by watching the set of symbols present in the problem, and adding the necessary type, cluster or other formulas for them, when certain criteria are met, proceeding in a fixpoint manner. Graphs are built from symbols to their background formulas before the fixpoint computation starts. The criteria for adding new background formulas are derived from close inspection of the Mizar checker's work with this information, which is quite a nontrivial matter[7]. However, the number of background formulas cut in this way from the problem is usually pretty high: the average

[7] Explanation of the internal workings of the Mizar checker and its use of the background information is beyond the scope of this article. The interested readers

size of an unfiltered theorem problem is about 570 formulas, which shrinks to about 73 formulas after filtering. This alone improves the provers' chances very significantly.

There are now several parameters to the filtering algorithm, allowing more or less restrictive versions, and also the interface to the filtering module is very simple, so that users can experiment with their own versions. This is perhaps not so important when reproving the MML theorems exactly (because then the necessary amount of filtering can be to great extent derived from the Mizar checker), but is useful for proving new theorems or finding new proofs for MML theorems.

The top-level problem creating script (Perl script mkproblem.pl) can be used to create both theorem and checker problems corresponding to the MML (reproving), as well as for creating new problems by specifying arbitrary MPTP formulas as axioms for proving another MPTP formula. The latter possibility can be used for all kinds of experiments, and e.g. the experiments with the Mizar Proof Advisor described in the Results section already make use of this possibility. The growing number of options guiding the problem generation, signature filtering, etc., is described in the mkproblem's manual page.

Even though all the theorem problems can take (depending on the filtering method, etc.) from 500 MB to about 3 GB, the problem generation is usually quite fast (5-15 minutes on 2GHz Intel Pentium 4), allowing fast rebuilding of the problems when experimenting with different options. Producing all checker problems takes about 10 GB.

Because of these sizes and the speed of problem generation, we decided not to distribute the generated problems with MPTP. This resembles a bit the approach used in the TPTP library, where only the generic format of problems is included (and indeed, users preferring other formats than DFG will have to perform similar problem translation work with tools like dfg2tptp or FLOTTER anyway). It is also motivated by the fact, that additional functionality is planned, that will increase the number of creatable problems even more.

2.5. The Database of Results

To facilitate the analysis of the results of provers run on MPTP, an experimental SQL (MySQL) database has been set up for them. The database is now restricted only to the theorem problems, mainly because of server limitations. Its detailed SQL structure is published at (MPTPResults). It now contains four tables: probleminfo, proved, proof and unproved. The probleminfo table contains information about the

can have a look at the source code of the filtering module, to get more on this, as well as at the articles (Wiedijk 2000) and (Naumowicz and Byliński 2002).

problems, independent of any prover runs. These are now entries like length of the Mizar proof, number of problem formulas, or info about the symbols occurring in the problem. The "proved" table contains statistics from successful provers' runs on the problems, while "unproved" contains unsuccessful runs. The "proof" table contains just the complete proofs corresponding to the "proved" table, and is kept separately from that table only because of its size and assumed limited use. A web interface to the database allowing arbitrary SQL selects is at `http://lipa.ms.mff.cuni.cz/phpMyAdmin-2.4.0` . This is now mainly used to look for suspicious spots in the translation, e.g. by comparing the length of a Mizar proof, with the length of the proof found by a theorem prover.

We would like to encourage MPTP users to contribute their results into the database, however, it is necessary to say that the structure of the database may still change a lot in the early versions. We will probably also have to find some "interestingness" criteria for including results into the database.

3. Problems, Limitations and Future Extensions

There are now several problems and limitations when using MPTP. Their up-to-date description and suggested workarounds are present in files README_MPTP.txt and MPTPFAQ.txt distributed with the system.

Several problems are caused by the infinite axiomatization (Tarski-Grothendieck set theory) used by Mizar, which leads to allowing second-order "schemes" in the language. The language also allows usage of the Fraenkel ("setof") operator. Because of the very restricted usage of the schemes, this can be solved in future versions by instantiating them, whenever they appear. Similarly, the Fraenkel terms can be "explained out" by adding axioms like:

```
x in {F(x1,...,xn): P[x1,...,xn]} iff
ex x1,...,xn st x = F(x1,...,xn) & P[x1,...,xn]
```

(together with the Extensionality Axiom, if it is not already part of the problem) which actually is exactly how the Mizar checker handles them.

We now do not keep track of the arithmetical evaluations (e.g. $6 = (8 + 10)/3$) performed by the Mizar checker. Some experiments have been done with using numeral encoding and some axioms of arithmetics to handle this, but it usually makes the problems much harder, with explosion of derivations of arithmetical facts. Our preferred way will be

to watch these inferences directly in the Mizar checker, and add them as axioms (e.g. $18 = 8 + 10$) whenever necessary. We see the long term solution of this problem in including efficient decision and evaluation procedures directly in ATP systems. However this is a nontrivial task.

Similarly, for the theorem problems, we need to get from the Mizar verifier information about its implicit unfolding of definitions inside proofs. Such unfoldings are controlled by the Mizar "*definitions*" environment directive, and the used definitions do not appear among the proof references. Again, adding all definitions made accessible by the "*definitions*"directive seems to make the problems significantly harder. Another solution is to try to use the signature filtering, to cut the space of all accessible definitions. Several options for this are already implemented in the development version of MPTP scripts.

Signature filtering seems to be working quite well now, especially for the checker problems. We use it for theorem problems too, however the current version may now in some cases be too strong (i.e. prune too much). The problem is, that we only use the external proof references, to create the initial symbol set that is used for the fixpoint computation. This is sound for checker problems, but there may be additional lemmas in the theorem problems, containing additional symbols, and causing that additional background formulas about those symbols might be needed. The solution that we plan to use for this, is to collect not only all external references appearing in proofs, but also all symbols present in the proofs, and use them as the initial symbol set for signature filtering. However, the estimate is, that approximating this by the symbols from external references should work well for most theorem problems in the first version.

Another problem is that on the contrary, the background theory can be too strong, because it is computed for full articles, and contains all background items introduced in them, which can possibly be used to justify some theorem coming in the article before them. This can be solved later e.g. by tagging all database items with their positions in Mizar articles.

It should be also noted that exact Mizar-like signature filtering is only important for exact reproving of MML theorems, which is not the only goal of the MPTP system (however important it is, e.g. for testing the quality of the translation). For proving theorems in new ways, or proving new theorems, efficient signature filtering over such a large library can, in most cases, only be heuristical, as it is a special version of the general problem of finding the most suitable lemmas from the library for proving an arbitrary new formula.

The probably largest future extension that we plan, is to export the structure of Mizar proofs too. This will allow all kinds of exper-

iments with lemma conjecturing or theory development. Right now, the library already makes it possible to do experiments with replacing theorem references with the references used in their proofs to arbitrary level, thus creating harder and harder problems. However, the problem generating scripts do not implement this option yet (though there is nothing difficult about it), and we may choose to wait with it until the full proof structure is available, and implement it more generally then.

Another line of development is to take into account, that most ATP systems do the main work on the clause level[8]. As of now, we have the integrity of the translated library on formula level, but skolem symbols are introduced during CNF translations done by provers, causing inconsistencies across various CNF problem inputs and the resulting proofs. That's why it would be good to have also a direct export of the library into CNF, introducing skolem symbols consistently.

There might be also some more experimenting with efficient encoding of the type information. We discuss this a bit more in (Urban 2003), where the inclusion-operator encoding of types suggested in (Dahn 1998) is also mentioned. A lot of experience in this area has been gained recently from the implementation of the MoMM project, where efficient (even though incomplete) type handling is crucial. However, as long as we are using SPASS with its efficient bottom-up sort mechanism, this matter is probably not pressing.

Finally, it would be nice to have more functions for comparing ATP proofs with Mizar proofs, or even some tools for at least semiautomatic translation of ATP proofs into Mizar. This would be useful for integrating well trained ATP systems as advice for Mizar authors. However, this is quite complex task, because of the very different proof formats, level of detail, and complicated structure of the Mizar language.

When speaking about all these possible improvements we should also note that the preferred goal is to have at least one theorem prover, that would be very much optimized for the Mizar problems (e.g. even by implementing some efficient Mizar-like type handling algorithms directly), because that might in turn boost the usability of the Mizar system and ease of formalization. That's why we will prefer to implement the features that go "in depth", rather than e.g. spending time on providing support for as many provers as possible. The reason for keeping the structure of the system simple, transparent and documented, is also to make it easy for others to cooperate, and to allow them to implement easily the features that they need.

[8] The OSCAR (Pollock 1996) prover (or rather AI system) being a notable exception from this rule, also the "lazy" clausification implemented in the Saturate system (Ganzinger and Stuber 2003) is interesting from this point of view.

4. First Results

MPTP 0.1 is based on MML 3.44.763, so all results refer to this version. There are 37617 theorem numbers in that MML. 4090 of them are canceled (i.e. unused in MML, but occupying the namespace, for the sake of continuity of the theorem numeration). So there are 33527 usable theorems in MML. Three of them are in fact set theory axioms from the article TARSKI, and hence without any proof. Proofs of 6078 of these theorems contain references that are not handled by MPTP 0.1 (either schemes or top-level non-theorem assertions in Mizar articles), so these problems are not eligible for reproving (though they are eligible e.g. for experimenting with finding new proofs). So for reproving, we are left with 27449 theorems (the 3 axioms are not worth taking special care of).

For all experiments we use the SPASS prover version 2.1, with 200 MB memory limit. The hardware is a cluster of computers with 700 MHz Intel Pentium-III processors running Debian GNU/Linux. Each problem is always assigned to a single processor

4.1. Reproving

4.1.1. *Reproving Filtered Problems*

The first basic experiment consisted in reproving the 27449 MML theorems, in as advantageous a setting as possible. This was done also in order to have some benchmark for other experiments.

We applied the default checker-based signature filtering, when creating the problems. Each problem was tried with 300s timelimit. The following table shows the results, and the figure shows how much time is needed to prove percentages of all the 11222 provable problems.

Table I. Experiment 1

proved	completion found	timeout	out of memory	unknown	total
11222	625	15149	352	101	27449

The average time for proving a provable problem is 14.12s. As the graph indicates, about 90 percent of all provable problems have been solved within 40 seconds, so after this "calibration" we decided to use for future experiments only 40s timelimit, to be able to conduct more of them. The overall CPU time needed for this full 300s experiment is about 70 days.

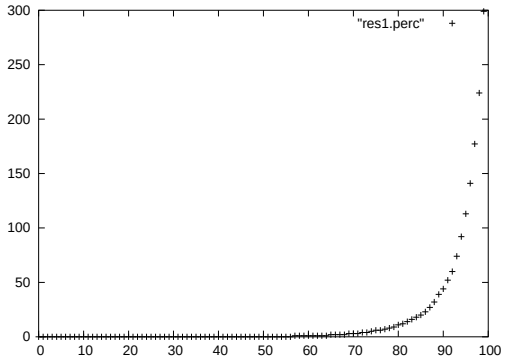

Figure 1. Time needed to solve percentage of problems in Experiment 1

The completions (i.e. those 625 problems for which SPASS found that the negated conjecture is not in contradiction with the axioms) can usually be explained by some of the reasons described in the Problems section - it can be lack of implicit definitions, or arithmetical evaluations, etc. On the other hand, some of the proved problems are sure to be proved in a MML-incorrect manner, e.g. as noted above, by having too strong background theory.

4.1.2. *Reproving Nonfiltered Problems*
To have some measure of how good the signature filtering is, we also try to prove the nonfiltered versions of the 27449 MML theorems. As noted above, only 40s timelimit is used, so comparisons should be done with $0.9 * 11222 = 10100$ proved problems from the previous experiment.

Again, the following table shows the results, and the figure shows how much time is needed to prove percentages of all the 3984 provable problems.

Table II. Experiment 2

proved	completion found	timeout	out of memory	unknown	total
3984	11	23447	2	16	27449

The average time for proving a provable problem is 6.54s. As already noted, the nonfiltered problems are much larger than the filtered versions, and within the same time, only about 40 percent of the amount for filtered versions is solved. Only for 191 problems it is the case, that the nonfiltered version was solved, while the filtered version not, so

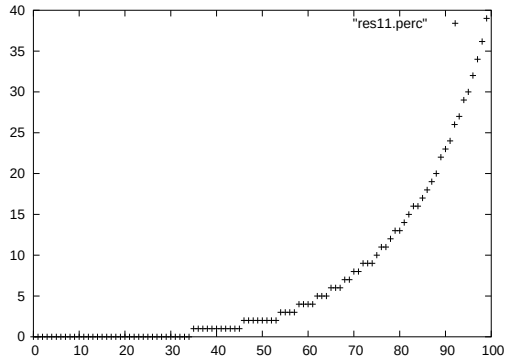

Figure 2. Time needed to solve percentage of problems in Experiment 2

running the nonfiltered versions is going to improve our knowledge in only about 2 percents of filtered-provable cases.

4.2. Mizar Proof Advisor

The first results are quite encouraging, especially if we realize that all are produced in a "push-button" manner, and there are still a lot of possibilities for improvement e.g. with optimizing ATP's performance by tuning many of their parameters, learning optimal orderings for various domains, etc.

Given these results, the logical next step is to really try to employ ATP systems, to assist Mizar authors with writing their formalized articles. To be able to do this, we have to turn attention to the very practical and pressing problem of choosing premises for a proof of an arbitrary formula, mentioned in the Motivation section.

One obvious answer to this problem, is to try using previous proof experience extracted from MML, to suggest a limited number of premises, that are most likely to be useful for proving an arbitrary formula. This idea is quite distant from the world of exact automated theorem proving, where completeness (though obviously very theoretical in the view of available resources) is one of the main issues, but we believe that it is an important aspect in humans' superiority over current ATPs when doing mathematics in large domains.

There are many possible machine learning and statistical approaches to the task of extracting and using proof experience from a corpus like MML, and some of them might even lead to interesting inductive/deductive architectures. However, for the beginning we decided to use a straightforward and well-known machine learning method,

for which tools are already available, and which could also serve as a benchmark for further more sophisticated approaches.

So the setting we chose for such a first attempt is following: We use a feature (attribute) based machine learning framework, in which the symbols (or rather constructors) present in formulas are the features that characterize them. This can be later improved e.g. by encoding parts of the formula structure as new features, or by switching to first-order learning systems. The output that we want from the system are MML theorems, ordered by their chance to be useful in the proof. This leads to the simple setting, in which there are many targets (MML theorems), characterized by their features (symbols occurring in them). Additionally, if theorem T, containing symbols $C_1, \ldots, C_n$ was in MML proved by theorems or definitions (shortly references) $R_1, \ldots, R_m$, we also want our system to notice, that not only T might be useful for proving something containing $C_1, \ldots, C_n$, but also its references $R_1, \ldots, R_m$ could be useful.

This idea might be recursively expanded (to include references of references, etc.), and further improved e.g. by using lower weights for more indirect references, but we postponed such tuning experiments for later time.

We have a very large number of features and targets, since there are about 40000 targets (references) and about 7000 features (constructors), and also quite a large number of training examples - about 33000 proved theorems. After some experimenting with various machine learning tools, we chose for the learning the SNoW (Sparse Network of Winnows) learning architecture (Carlson et all 1999), used mainly for natural language processing tasks. It is designed to work efficiently in such tasks, where the number of features and targets is very large. SNoW implements several learning algorithms, from which the naive Bayes seems to work best on our data.

The option to run the trained system in a server mode is already implemented in SNoW, so regardless of the theorem proving experiments, it was very easy to set up a server that is already now providing hints to Mizar authors.

4.2.1. *Evaluation of the Advisor*

In the first experiment, we used the standard 10-fold cross-validation, to test the prediction capability of SNoW trained on our data. The 33527 examples were randomly split into 10 equally large sets, and in 10 runs, SNoW was trained on the 9 sets and evaluated against the missing set.

In the testing mode, SNoW outputs for each example the list of hints (references), that it evaluates as useful for the example, ordered

by their expected utility. We are interested in how good such predictions are, and in this case, this is measured by looking at the example's real references, and counting their ratio among the hints given by SNoW, as the hint limit is increased. This is measured on the scale ranging from 1 to 100 hints. We decided to modify this ratio at the beginning, so that if we e.g. only require one hint, and that hint is correct (i.e. it is among the real references), the success ratio is 1 instead of 1 / (number of real references), which we think corresponds more closely to the intuitive idea of "success" of the prediction. The number of real references is usually much lower than 100 (about 10 on average), so this modification effects only the very beginning of the scale, and at numbers larger than 20, it is already quite correct to interpret the ratio also as the coverage of the real references among the SNoW hints. The following graph shows this value, averaged across the 10 leave-one-out SNoW evaluations.

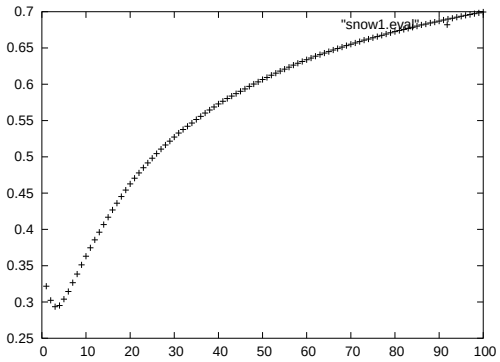

Figure 3. Ratio of necessary references in SNoW hints

The drop at the beginning of the graph is caused exactly by our modified definition of "success". The final value of 0.7 for 100 hints means that on previously unknown formulas, about 70 percent of references needed for their proof will be present among the first 100 hints.

We should note, that *needed references* is not exactly correct expression here, the exact meaning is *references used for the MML proof*. It is possible that the SNoW hints will lead to an alternative proof of the formula, and actually, we plan to do experiments exactly with the purpose to use this difference to try to find alternative proofs for MML theorems.

It could be suggested, that even though we do the testing on unknown data, the 10-fold cross-validation does not exactly correspond

to the setting in which SNoW will be used most often. The typical situation is, that all of MML is already known, and the author is writing a new article, that will be in the end appended to MML. In more detail, it would be possible to train SNoW also on the theorems proved so far in the article written by the author.

That's why we run another evaluation, corresponding to this setting. SNoW is incrementally trained on the MML examples in the MML processing order, each time having complete information about the preceding articles and theorems, and having no information about the subsequent articles and theorems. Obviously, in this setting, it only has sense to ask from the SNoW the proof references of the tested theorem as hints, the theorem itself cannot be hinted, as it was not yet seen by the system. The following graph shows results of such evaluation.

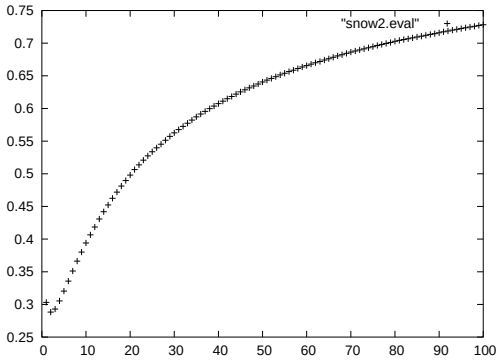

Figure 4. Ratio of necessary references in SNoW hints 2

It is also interesting to see how the ratio changes as the MML grows, which is for the limit value 100 shown on the next graphs. Since the discrete graphs ranging across all 33000 MML theorems are poorly readable, the first graph applies smoothing across the previous 1000 values, and the second applies smoothing across the previous 100 values.

4.3. Proving New Theorems Using the Advisor

Having prepared both MPTP and the Proof Advisor, we can finally conduct the experiment with proving previously unknown formulas. For that, we use the hints provided by the incremental training and testing on growing MML, described in the previous section. As noted, it means that at each point, the theorem we are trying to prove is the most recent addition to the MML, it was never seen before nor could be used as a reference for another theorem.

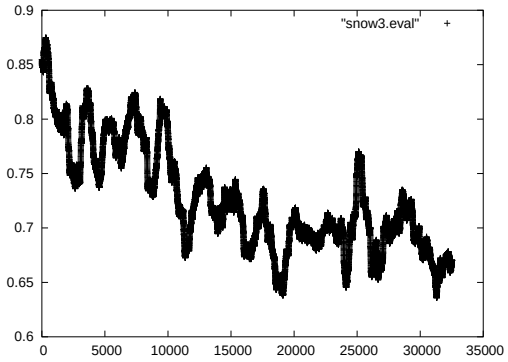

Figure 5. Hint ratio for limit 100 as the MML grows, smoothing = 1000

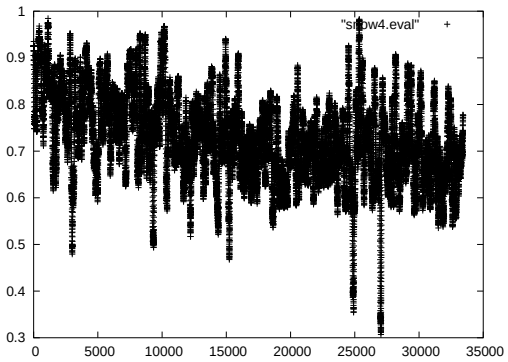

Figure 6. Hint ratio for limit 100 as the MML grows, smoothing = 100

We have to choose our policy for selecting the number of hints for constructing the proof problems. Choosing too many creates the danger of "suffocating" the prover, like in the case of using the nonfiltered background knowledge, and choosing too few leads into the risk of incompleteness. Looking at the hint ratio graph from the incremental training, we decided to use for this first experiment the value of 30 hints, which on average guarantees about 50 percent of the original MML proof references. Since the SNoW system also outputs the prediction strengths, when it evaluates the targets, maybe a more reasonable option for future experiments could be selecting the number of hints selectively, according to their prediction strength, rather than uniformly with one limit for all.

We apply the standard signature filtering to the problems, although as noted, it is (mostly) guaranteed to preserve completeness only for reproval tasks. The Advisor-like approach could be in the future ex-

tended to handle also background creation. All problems are again run with 40s timelimit, all other settings being same as for the reproval experiments, with the exception of the number of problems attempted - unlike in reproval, where problems with unexported references were discarded, we attempted all 33527 MML problems, since SNoW only provides valid references as hints. The results are shown in the following table.

Table III. Experiment 7

proved	completion found	timeout	out of memory	unknown	total
4825	7	28580	69	46	33527

So even with very straightforward implementation of the Proof Advisor, various completeness issues still involved in the first version of MPTP, and some quite arbitrary choices, like the number of hints used, and practically no ATP optimizations, it is already possible to automatically prove within 40 seconds about every seventh newly attempted Mizar theorem.

It might be argued that there are more and less difficult theorems in MML, and that e.g. in terms of number of lines written by Mizar authors, proofs of these theorems will probably be shorter, so the amount of work saved to Mizar authors will be less than this ratio. The obvious answer is that ATPs can be in this mode applied to any Mizar formula, not just top-level theorems, so even the proofs of hard theorems can thus be made significantly simpler for the formalizers, by applying ATPs to the lemmas that occur during writing the proofs. As noted, we hope to include the full proof structure into the next MPTP release, which will enable us to quantify, to what extent this really applies to the current MML. Finally, to put these results into a proper context, note that in (Wiedijk 2002), the cost of creating the MML library is estimated to about 90 man-years.

5. The Second Goal of This Article

The main goal of this article was to provide a description of the MPTP system and an overview of its first results. However, there is also a secondary goal. We wanted to persuade the readers (especially those working in ATP and formalization projects), that cooperation between ATP systems and large formalization efforts is useful for them.

This may seem trivial, but author's experience gained during implementing MPTP is, that it is not, and a considerable part of such effort is spent on persuading. Even with essentially first-order systems like Mizar, there are number of features that make the translation to ATP formats and efficient use of ATP systems on translated problems difficult. Even though ATP-friendly implementation of such features is in most cases possible, it usually has low priority, as the formalization people usually look quite skeptically at the possibility of having some real benefits from ATP systems. Similarly, ATP systems today are quite firmly seated in the simple unrepeated-axioms-conjecture paradigm for problems formulation and solving. It is frowned upon, if an ATP system has some built-in domain optimizations, and it is even difficult to do trivial changes to common ATP input formats, that would allow to express previous knowledge to hint the provers. It is good to have provers that perform well on artificial problems, but it is even better to have domain-optimized provers being of some use in real mathematics.

We hope that the presented experimental results already show, that the cooperation is useful, and worth supporting by more than just words.

6. Acknowledgments

As already noted, a lot of ideas come from the previous integration of MML into the ILF project. Thanks to Ingo Dahn, Christoph Wernhard and Czesław Byliński for making that work available to me. Thanks to the team developing Mizar, naming at least Andrzej Trybulec, for continuous in-depth discussions of the system. Finally thanks to my supervisor, prof. Petr Štěpánek, for more general advice when dealing with various problems, and continuous support.

The final version of this article contains numerous improvements suggested by the anonymous referees. I appreciate very much both their effort to improve the quality of this article, and their suggestions and comments on the MPTP system.

References

Bancerek G. [2000], Development of the Theory of Continuous Lattices in Mizar. In M. Kerber and M. Kohlhase, editors, Symbolic Computation and Automated Reasoning, pages 65-80, The CALCULEMUS-2000 Symposium, A.K.Peters, Natick, Massachusetts.

Carlson A. J., Cumby C. M., Rosen J. L. and Roth D. [1999], SNoW User's Guide. UIUC Tech report UIUC-DCS-R-99-210.

Dahn I. and Wernhard C. [1997], First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. Proceedings of the International Workshop on First order Theorem Proving. RISC-Linz Report Series, No. 97-50, Johannes Kepler Universität Linz,

Dahn I. [1998], Interpretation of a Mizar-like Logic in First Order Logic. Proceedings of FTP 1998. pp. 137-151.

Dahn I, [2001], Slicing Book Technology - Providing Online Support for Textbooks, Proc. ICDE 2001, International Conference on Distant Education, Düsseldorf.

Ganzinger H., Meyer C. and Weidenbach C. [1997], Soft Typing for Ordered Resolution. In Proc. CADE-14, pp. 321-335, Springer.

Ganzinger H. and Stuber J. [2003], Superposition with Equivalence Reasoning and Delayed Clause Normal Form Transformation. In Proc. 19th Int. Conf. on Automated Deduction (CADE-19), Miami, USA. Springer LNAI 2741, pages 335-349.

Hähnle R., Kerber M. and Weidenbach C. [1996], Common Syntax of the DFGSchwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany.

Lenat D. B. [1979], On automated scientific theory formation: A case study using the AM program. In J. Hayes, D. Michie and L.I. Mikulich (Eds.), Machine Intelligence 9 (Halstead, New York, 1979) 251-283.

Lenat, D. B. [1982] The nature of heuristics. In Artificial Intelligence, Vol. 19, No. 2, October 1982, North-Holland, Amsterdam.

McCune, W. W. [1997], Solution of the Robbins Problem. Journal of Automated Reasoning 19(3), 263–276, 1997.

The MoMM interreduction system by Josef Urban, available online at `http://alioth.uwb.edu.pl/twiki/bin/view/Mizar/MoMM`.

MPTPResults.sql - SQL structure of the MPTP result database, published online at `http://alioth.uwb.edu.pl/twiki/bin/view/Mizar/MpTP`.

Naumowicz A. and Byliński C. [2002], Basic Elements of Computer Algebra in MIZAR, Mechanized Mathematics and Its Applications Vol. 2(1), August 2002.

Nonnengart A. and Weidenbach C. [2001], Computing small clause normal forms. In A. Robinson and A. Voronkov, eds., Handbook of Automated Reasoning, Vol. 1, Elsevier, chapter 6, pp. 335-367.

Pollock J. L. [1996], OSCAR: a general purpose defeasible reasoner, Journal of Applied Non-Classical Logics 6, (1996), 89-113.

Rudnicki P. [1992], An Overview of the Mizar Project, Proceedings of the 1992 Workshop on Types for Proofs and Programs, Chalmers University of Technology, Bastad.

Schulz S. [2002], E – A Brainiac Theorem Prover, Journal of AI Communications, Vol. 15, pp. 111-126.

Schulz S. [2001], Learning Search Control Knowledge for Equational Theorem Proving, In F. Baader and G. Brewka and T. Eiter (Eds.), Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI-2001), LNAI Vol. 2174, pp. 320–334, Springer.

Schumann J. M. [2001], Automated Theorem-Proving in Software Engineering. Springer-Verlag.

Sutcliffe G. and Suttner C.B. [1998], The TPTP Problem Library: CNF Release v1.2.1, Journal of Automated Reasoning, Vol. 21/2, pp. 177-203.

Sutcliffe G., Zimmer J. and Schulz S. [2003] Communication Formalisms for Automated Theorem Proving Tools, In V. Sorge and S. Colton and M. Fisher and J. Gow (Eds.), Proceedings of the IJCAI-18 Workshop on Agents and Automated Reasoning, pp. 53–58.

Urban J. [2003], Translating Mizar for First Order Theorem Provers. In Andrea Asperti, Bruno Buchberger, James Davenport (eds.), Mathematical Knowledge Management, Proceedings of MKM 2003, LNCS 2594.

Urban J. [2004], MoMM - Fast Interreduction and Retrieval in Large Libraries of Formalized Mathematics. Accepted to editors Geoff Sutcliffe, Stefan Schultz and Tanel Tammit - Proceedings of the IJCAR 2004 Workshop on Empirically Successful First Order Reasoning, ENTCS. Available online at `http://ktiml.mff.cuni.cz/~urban/MoMM/momm.ps`

Weidenbach C. [2001], SPASS: Combining Superposition, Sorts and Splitting, in A. Robinson and A. Voronkov (Eds.), Handbook of Automated Reasoning, Vol II, Chapter 27, pp. 1965–2013, Elsevier Science and MIT Press.

Wiedijk F. [2000], CHECKER - notes on the basic inference step in Mizar. available at `http://www.cs.kun.nl/~freek/mizar/by.dvi`

Wiedijk F. [2002], Estimating the Cost of a Standard Library for a Mathematical Proof Checker. `http://www.cs.kun.nl/~freek/notes`.

Wiedijk F. [2003] Comparing mathematical provers, In Andrea Asperti, Bruno Buchberger, James Davenport (eds.), Mathematical Knowledge Management, Proceedings of MKM 2003, LNCS 2594, pp. 188-202.

Translating Mizar for First Order Theorem Provers

Josef Urban
urban@kti.ms.mff.cuni.cz
Dept. of Theoretical Computer Science
Charles University
Malostranske nam. 25, Praha, Czech Republic

July 27, 2004

Abstract

The constructor system of the Mizar proof checking system is explained here on examples from Mizar articles, and its translation to untyped first-order syntax is described and discussed. This makes the currently largest library of formalized mathematics available to first-order theorem provers.

1 Introduction, Previous Work

Mizar [Rudnicky 92] is a system for computer checked mathematics. In more detail, Mizar is associated with several things:

- The Mizar language ... this is the language in which Mizar articles must be written, so that they can be checked by computer.
- The Mizar Mathematical Library (MML). This is the growing (now about 700) collection of Mizar articles that have already been written and computer checked and the notation, definitions, theorems and other Mizar constructs created in them can be used for writing new Mizar articles. Various presentations of the MML exist today: Formalized Mathematics, online html-ized abstracts, Mizar Encyclopedia.

- The Mizar checker and other software utilities for working with articles and MML.
- The Mizar project headed by A.Trybulec, taking care of the things named above as well as other things related to Mizar.

Several introductions to the Mizar language as well as to the practical aspects of writing Mizar articles exist today e.g.
[Bonarska 90, Muzalewski 93, Rudnicky 92, Wiedijk 99]. Although it is much recommended to have a look at one of these introductions, we will try here to explain the features of Mizar, which are relevant for the first order translation.

The probably largest effort at translating the MML to first order syntax was carried out several years ago by the ILF group [Dahn 97]. The work described here is to a great extent inspired by the previous ILF work, and many ideas used here originate from there. We give a short description of it, as well as its relationship to the present work at the end of this article.

2 General Descriptions

2.1 The Mizar Language

The detailed specification of the Mizar language is given in [Syntax]. For the reader's convenience, text version of this specification is given in Appendix A. We will loosely follow this specification here, notions that are exactly defined by this syntax will be written in italics here.

The main parts of a Mizar article are definitions (*Definitional-Item*s), *Theorem*s and *Scheme*s. It is possible to define functions (*Functor-Definition*), predicates (*Predicate-Definition*), types (*Mode-Definition, Structure-Definition*) and attributes (*Attribute-Definition*).

Attributes are unary predicates handled in a special way by the system. The rules of attribute handling are defined by the user in *Cluster-Registration*s.

To exemplify the abstract syntax, we show here the beginning of the basic article about real numbers (real_1.miz):

```
:: Basic Properties of Real Numbers
::  by Krzysztof Hryniewiecki
::
:: Received January 8, 1989
:: Copyright (c) 1990 Association of Mizar Users

environ

 vocabulary REAL_1, NUMBER;
 constructors ARYTM;
 requirements ARYTM, SUBSET;
 notation ARYTM, SUBSET_1;
 clusters ARYTM, ARYTM_3;
 definitions TARSKI, ARYTM;
 theorems ARYTM, AXIOMS;
 schemes BOOLE;

begin

definition
  mode Real is Element of REAL;
end;

reserve r       for set;
reserve x,y,z,t,a,b,c,d for real number;

:: Basic properties of '+', '*'
canceled 8;

theorem
Th9:
 z<>0 & x*z=y*z implies x=y
proof
  assume  z<>0;
     then consider z' being real number such that
       A1: z * z' = 1 by AXIOMS:20;
     assume x * z = y * z;
       then x * (z * z') = y * z * z' by AXIOMS:16;
```

```
       then x = y * 1 by AXIOMS:16,A1;
       hence thesis;
end;

theorem Th10:
   x + z = y + z implies x=y
proof
        consider z' being real number such that
          A1: z + z' = 0 by AXIOMS:19;
          assume x + z = y + z;
          then x + (z + z') = y + z + z' by AXIOMS:13;
          then x = y + 0 by AXIOMS:13,A1;
          hence thesis;
end;

definition let x be real number;
  func -x -> real number means :Def1: x + it = 0;
  existence by AXIOMS:19;
  uniqueness by Th10;

  assume A1: x<>0;
  func x" -> real number means :Def2:  x * it = 1;
  existence by AXIOMS:20,A1;
  uniqueness by Th9,A1;
end;

definition let x,y be real number;
  func x-y equals :Def3: x+(-y); correctness;

  func x/y equals :Def4: x * y"; correctness;
end;

definition let x,y be real number;
  cluster x-y -> real;
coherence
  proof
    x-y = x+-y by Def3;
    hence x-y  REAL by ARYTM:def 2;
```

```
  end;
  cluster x/y -> real;
coherence
  proof
    x/y = x*y" by Def4;
    hence x/y  REAL by ARYTM:def 2;
  end;
end;
```

The article starts with some commented bibliographic information, after that follows *Environment-Declaration* (*Directive*s saying which parts of the MML are used by this article). The *Text-Proper* part of the article starts with *Mode-Definition* of the mode Real, after that are two *Reservation*s, then two *Theorem*s with *Proof*, and finally three *Definitional-Items*. The first *Definitional-Item* defines two unary functions ("-" and """) , the second defines two binary functions ("-" and "/") and the third consists of two *Cluster-Registration*s, which add the attribute "`real`' to the result of applying functions "-" and "/".

All Mizar terms are typed. There is a largest (default) type called "`set`" or "`Any`". All other types have one or more mother types. Types of variables are given either in global *Reservation*s or local *Loci-Declaration*s or inside quantified formulas. Types of other terms are computed according to *Functor-Definition*s. Types can have arguments (be parameterized) in Mizar, e.g. "`Element of X`" or "`Function of NAT, REAL`" are legal types with one or two arguments, respectively.

Type translation is the largest part of the first order translation. There are two possible basic approaches to type translation:

- Types can be thought of as set-theoretic classes (e.g. type "`set`" being the universal class, type "`Element of X`" being the set of all elements of X, type "`Integer`" being the set of all integers, etc.).
- Types can be thought of as predicates .... thus "`set(X)`" is true for any X, "`Integer(X)`" is true iff X is integer, and "`Element(X,Y)`" is true iff $X \in Y$

Both approaches have some advantage, we use the second (predicate) approach, mainly because some provers (e.g. SPASS [Weidenbach et all 99] have optimizations available for monadic predicates, to which all the zero-argument types translate.

In [Dahn 98] another translation of types based on the idea of inclusion operations ([Goguen 92]) is suggested. This translation has the potential to handle single inheritance type hierarchies efficiently, however this efficiency is lost in the case of Mizar attributes, which are a very large part of the Mizar type system. We hope that it might be possible to combine this kind of translation for the single inheritance part of Mizar with the efficient optimizations done by SPASS in the future.

2.2 Syntactic Levels

The Mizar language handles a very large database of articles about different parts of mathematics and this necessarily leads to notation conflicts. The solution to such conflicts is introduction of two syntactic levels of the language: the **pattern** level and the **constructor** level.

Constructors are the real unambiguous functions, predicates, types and attributes to which the patterns are translated before any proof checking takes place. Patterns are mapped to the constructors, they accommodate the need for having different symbols for the same constructor or vice versa (same symbol for different constructors).

The process of mapping patterns to constructors is done separately for each Mizar article depending on its *Environment-Declarations* and is usually quite nontrivial.

Example: binary symbols "`in`" and "`<‘`" define different patterns, but when dealing with ordinals, they are mapped to the same constructor, i.e. "`A in B`" and "`A <‘ B`" cannot be distinguished when translated to the constructor level.

Definitions can influence both the pattern and the constructor level. Typically, a definition causes a new pattern and also a new constructor to be defined, but in many cases there is no effect on the constructor level.

For the purpose of translating Mizar articles into first order syntax, only the constructor level is important, the patterns can be thought of as a "syn-

tactic sugar" added on top of the constructors.

2.3 First Order Formats, Outline of the Translation

Several kinds of first order syntax are used today for first-order provers, e.g. TPTP format [Suttner and Sutcliffe 98], DFG format [Hahnle et all 96], Otter format [McCune 94] and others. We chose direct translation into DFG format, since our immediate purpose was to experiment with the SPASS prover. More generic approach is certainly desirable, however, at a first glance, several such approaches come into mind (e.g. ILF-like approach [Dahn 97], TPTP-like approach, or even adding direct Mizar support for other formats than just DFG), so we postponed such decisions for later time. Note that the dfg2tptp tool can be used for translation to TPTP and thus (using TPTP tools) to virtually any other syntax.

So the general approach to translation is following:

- We use parts of Mizar (Parser, Analyzer) which translate the Mizar articles to the constructor level, where our first order translation starts.
- We give absolute (context independent) names to all constructors[1].
- Definitions of constructors usually translate to several first order formulas, since we have to translate both the type hierarchy information given in the definitions and the actual *Definiens*.
- Sometimes also additional properties (e.g. commutativity, transitivity, etc.) of the defined constructors are stated inside definitions. They are also translated into corresponding first order formulas.
- All formulas are relativized with respect to the typed variables occurring in them (using the above mentioned predicate translation of types). So e.g. universally quantified Mizar formula

  "`for x being Real holds x-x = 0`" translates to first order formula

  "`Real(x) implies x-x = 0`".

[1]The naming scheme we use in what follows has obviously no importance for theorem provers, however, it has become standard in various Mizar presentations, allowing their interoperability.

Remark: The actual DFG syntax translation of the above mentioned Mizar formula is:

```
forall([B1], implies(v1_arytm(B1), equal(k3_real_1(B1,B1),0))).
```

However, for explanation purposes, we use here rather the user-defined symbols instead of the absolute constructor names, and also for the sake of better readability, we do not strictly adhere to the DFG syntax of formulas.

Next we give a detailed explanation of the translation for all kinds of Mizar definitions.

3 Translation of Mizar Definitions

3.1 Mode Definitions

Types in Mizar can be defined using either *Mode-Definitions* or *Structure-Definitions*. We deal with modes first. Since structures are more complicated, we postpone them after the explanation of functions.

As already stated, the translation of the largest (default) type "`set`" is a predicate that always holds true ("`set(X)`" is true for all terms X). So omitting relativization by this predicate is logically correct, and we do it for the sake of better readability of translated formulas.

For all other modes, the syntactic structure of *Mode-Definition*s is:

`mode` *Mode-Pattern* `(` `[` *Specification* `]` `[` `means` *Definiens* `]`
Correctness-Conditions `;` `|` `is` *Type-Expression* `;` `)` { *Mode-Synonym* } .

Example:

```
definition
  mode Real is Element of REAL;
end;
```

This is the first definition from `real_1`, defining the mode "`Real`" as a shorthand for the (already known) mode "`Element of REAL`" (we give the definition of the mode "`Element`" as the next example, the constant "`REAL`"

is defined in the article `arytm.miz` and has the meaning of the set of all real numbers).

The *Mode-Definition*s that use the keyword "`is`" are called "expandable" definitions in Mizar. Such definitions do not introduce a new constructor, they merely create a new pattern (given here by the symbol "`Real`" and arity 0) and map it to already known constructors (here the constructors corresponding to the patterns "`Element`" and "`REAL`", i.e. "`m1_subset_1`" and "`k1_arytm`" in absolute notation). Since our translation works directly with the constructor level, there is no work to be done for such "expandable" definitions.

Example:

```
definition let X;
 mode Element of X means
:Def2: it in X if X is non empty
   otherwise it is empty;
 existence by BOOLE:def 1;
 consistency;
end;
```

This is a definition of the mode "`Element of X`" from the article `subset_1.miz`. The *Loci-Declaration* "`let X;`" declares the variables occurring in the definition. Since type is not given for "X", it defaults to the largest type "`set`".

This definition omits the optional *Specification* part. If the *Specification* is present in the definition, it gives a mother type for the newly defined mode. If not, the largest type "`set`" is used as the mother type, i.e.

"`mode Element of X means ...`" has the same effect as
"`mode Element of X -> set means ...`" .

The *Definiens* after the keyword "`means`" consists of an optional *Label-Identifier* and either a simple sentence or several sentences separated by the keywords "`if`" and "`otherwise`". The latter (*Conditional-Definiens*) is used for "per cases" definitions.

In our example, the *Conditional-Definiens* distinguishes the case when the mode argument X is non empty from the case when the mode argument X is empty. The sentence for the first case states, that for any "`it`" (special

variable used in definitions) being of the type "`Element of X`" means "`it in X`". The second case sentence says, that in the degenerate case when X is empty, being of the type "`Element of X`" means to be empty too.

If the reader is curious about why the degenerate case is handled this way, the answer is, that simply in many cases it turned out to be advantageous.

The keywords "`existence`" and "`consistency`" in the example introduce *Correctness-Conditions*. The "`existence`" condition is compulsory for all non-expandable modes, it states that the extension of the defined type is non empty (here it is proved by referring to the definition 1 in article `BOOLE`).

The "`consistency`" condition states that the disjunction of all the cases into which the definition was split, is true (it is trivial here).

The *Mode-Synonym* is not used in our example. If given, it just defines another pattern mapped to the same constructor. In this example, the definition creates both a new pattern (associated with the symbol "`Element`") and a new constructor (to which the pattern is mapped).

We create an absolute name for the constructor as follows:

The first letter denotes the kind of the constructor ... it is "`m`" for modes. After that follows its article relative number, i.e. since this is the first mode constructor in article `subset_1`, the number is 1. After that we append the Mizar identifier of the article ... it is `subset_1`. So the absolute name is "`m1_subset_1`". This absolute naming is used quite often in various Mizar presentations.

We translate types as predicates, so n-ary types will become n+1-ary predicates, and e.g. the Mizar type qualification "`X is Element of Y`" will be translated as "`m1_subset_1(X,Y)`".

Next we want to encode the part of the type hierarchy created by this definition. As noted above, the mother type of "`Element of X`" is the largest type "`set`". The translation would be following:

```
‘‘forall([X,Y], m1_subset_1(Y,X) implies set(Y)).’’
```

However, as already mentioned, "`set(Y)`" is always true and we chose not to translate it, so in such cases (i.e. when the mother type is "`set`"), we do not create the type hierarchy translation.

We should mention that such formulas have to be normally relativized by types of the variables occurring in them, so it would be:

```
‘‘forall([X,Y], set(X) implies (m1_subset_1(Y,X) implies set(Y))).’’
```

But the “pruning” of the type “`set`” would apply here too. This silent pruning will be done in all following formulas.

Translating the definiens is straightforward. We translate the definition formula for each case and conditionalize by the case assumptions. So the first case (when “`not(empty(X))`” holds) translates to:

```
‘‘m1_subset_1(Y,X) iff in(Y,X)’’
```

Similarly, the second case (when “`empty(X)`” holds) translates to:

```
‘‘m1_subset_1(Y,X) iff empty(Y)‘‘
```

after conditionalization:

```
‘‘m1_subset_1(Y,X) iff ( ( not(empty(X)) and in(Y,X) )
or ( empty(X) and empty(Y) ) )’’
```

(instead of “`empty`” and “`in`” their absolute names would be used).

Finally, we translate the existence condition:

```
‘‘forall([X], exists([Y], m1_subset_1(Y,X))).’’
```

Mizar allows for explicit typing of terms using the (heavily overloaded) keyword “`is`”. E.g.:

“`for i being Integer holds (i > 0 implies i is Nat)`”

is a legal Mizar sentence explicitly typing positive integers to natural numbers.

Since we already translate types as predicates, there is no need to translate the Mizar typing predicate “`is`”, and e.g. the atom “`i is Nat`” is translated directly as “`Nat(i)`”.

3.2 Predicate Definitions

In comparison to first order predicates, Mizar predicates can have several predefined properties (e.g. reflexivity, symmetry, etc.), and they also define the type of their arguments. *Predicate-Definition* has following syntactic structure:

`pred` *Predicate-Pattern* `[ means` *Definiens* `] ;` *Correctness-Conditions*
{ *Predicate-Property* } { *Predicate-Synonym* } .

```
e.g.:
 definition let X,Y;
  pred X c= Y means
:: TARSKI:def 3
   x in X implies x in Y;
  reflexivity;
 end;
```

This is a definition of the subset relation from article `tarski`. Again first comes a *Loci-Declaration*
"`let X,Y;`", after that the *Predicate-Pattern*
"`X c= Y`", then the *Definiens*
"`x in X implies x in Y;`" and finally one *Predicate-Property* is stated:
"`reflexivity;`".

*Correctness-Condition*s occur only in redefinitions of predicates, redefinitions will be discussed later.

Again, this definition creates both a pattern and a constructor, the standard symbol for predicate constructors is "`r`", so the absolute name would be "`r1_tarski`" here.

The *Definiens* formula is simply translated as equivalence:
```‘‘r1_tarski(X,Y) iff (x in X implies x in Y)’’```

Predicate-Property can be:
`symmetry connectedness reflexivity irreflexivity`
So we add the corresponding theorem for such properties, here it is:
```‘‘r1_tarski(X,X).’’```

Pruning of the "`set`" relativization took place in the previous two formulas, without the pruning the translation would be:
```‘‘(set(X) and set(Y)) implies (r1_tarski(X,Y) iff (x in X implies x in Y))’’``` and
```‘‘set(X) implies r1_tarski(X,X).’’```

3.3 Attribute Definitions

As mentioned, attributes are unary[2] predicates handled in a special way by the system. The *Cluster-Registration*s, which define rules of attribute handling will be explained later.

The syntactic structure of *Attribute-Definition* is:
`attr` *Attribute-Pattern* `means` *Definiens* `;` `[` *Correctness-Condition* `]` { *Attribute-Synonym* } .

Example:

```
definition let M;
 attr M is limit means
:Def7:  not ex N st M = nextcard N;
  synonym M is_limit_cardinal;
end;
```

This is a definition of the attribute "`limit`" from article `card_1` (saying that a cardinal is limit iff it is not a successor of another cardinal). The variables M and N occurring in the definition were earlier in the article globally reserved for type "`Cardinal`":
"`reserve K,M,N for Cardinal;`"

Since attributes are in fact specially treated predicates, the syntax of the definition is very similar to *Predicate-Definition*s. The only difference is the usage of the keyword "`is`" in the *Attribute-Pattern*:
"`M is limit`".

This definition uses the optional *Attribute-Synonym* slot:
"`synonym M is_limit_cardinal;`", which adds another pattern.

So this definition creates two new patterns (bound to symbols "`limit`" and "`is_limit_cardinal`") and maps both of them to one newly created constructor. The standard symbol for attribute constructors is "`v`", so its absolute name is "`v1_card_1`".

[2]This is true only on the syntactic level. If the type of the attribute's argument depends on a variable, this variable also becomes an argument of the attribute. There are plans to allow attributes with multiple arguments even on the syntactic level now.

The *Definiens* is again translated as equivalence:
‘‘`v1_card_1(X) iff not( exists([Y], equal(X, nextcard(Y))))`’’.

Since the types of variables are nontrivial in this definition, we have to relativize the formula:
‘‘`Cardinal(X) implies ( v1_card_1(X) iff not( exists([Y], Cardinal(Y) and equal(X, nextcard(Y)))))`’’

A possibly negated attribute is called *Adjective* in Mizar. A finite set of *Adjective*s is called *Adjective-Cluster* (or just cluster) in Mizar. Clusters can be used as prefixes to types, e.g. “`finite non empty set`” uses the *Adjective*s “`finite`” and “`non empty`” as a prefix to the type “`set`”.

Such types with non empty *Adjective-Cluster* are translated as a conjunction of all the predicates corresponding to the attributes and the predicate corresponding to the underlying type, so here it is:
“`finite(X) and not(empty(X)) and set(X)`” for a variable X of the type “`finite non empty set`”.

Again, the “`set`” can be pruned, so the translations is just ‘‘`finite(X) and not(empty(X))`’’.

3.4 Functor Definitions

Functions in Mizar have to define the types of their arguments and the type of their result. Several *Functor-Properties* (e.g. “`commutativity`”) can be associated with them.

The syntax of *Functor-Definition* is:
`func` *Functor-Pattern* `[` *Specification* `]` `[` `(` `means` | `equals` `)` *Definiens* `]` `;` *Correctness-Conditions* { *Functor-Property* } { *Functor-Synonym* } .

Example:

```
definition let x be real number;
  func -x -> real number means :Def1: x + it = 0;
  existence by AXIOMS:19;
  uniqueness by Th10;
```

This is the definition of the unary function “`-`” from article “`real_1`”, already mentioned above. The initial *Loci-Declaration* declares a variable x for the type “`number`” with *Adjective* “`real`”.

Type "number" is now defined just as a convenient synonym for the largest type "set" (in article arytm), "real" is an attribute defined also in article arytm. The *Functor-Pattern* is
"-x" here, and the *Specification*
"-> real number" defines the result type of the function. The *Definiens*
":Def1: x + it = 0;" is the definitional formula of the defined function.

The *Correctness-Condition* "existence" states, that there exists an object (of the desired type "real number") conforming to the *Definiens* (here it is proved by reference to the theorem 19 in article axioms).

"uniqueness" says that such object is unique (proved by previous theorem Th10). Before accepting a new *Functor-Definition*, the system always checks these two conditions.

This definition creates one new pattern and one new constructor. The standard Mizar symbol for functor constructors is "k" so the constructor's absolute name is ''k1_real_1'' (first functor in article real_1). We need to translate the typing, the *Definiens* and possibly the *Functor-Properties* (none in this case).

The typing translates to:
"'real number'(X) implies 'real number'(k1_real_1(X))."

Following the note about *Adjective-Cluster*s, 'real number'(X) translates to
"(real(X) and number(X))"
and since "number" is just a synonym for "set" it is pruned to just "real(X)". Since "real"'s absolute name is "v1_arytm", the exact translation is:
"forall([X], implies(v1_arytm(X), v1_arytm(k1_real_1(X))))".

The *Definiens* is translated by first instantiating the special variable "it" with the *Functor-Pattern*, yielding:
"x + (-x) = 0;" and translating the result formula, which gives:
"equal(+(x,k1_real_1(x)),0)".
After relativization and replacing "+" with its absolute name "k3_arytm", we get:
"forall([X], implies(v1_arytm(X), equal(k3_arytm(X,k1_real_1(X)),0)))"

Here is another example of *Functor-Definition* from the article real_1:

```
definition let x,y be real number;
  func x-y equals :Def3: x+(-y); correctness;
```

This differs from the previous definition in using the keyword "`equals`" instead of the keyword "`means`". A binary function "`-`" operating on real numbers is defined here. The *Definiens* "`x+(-y);`" is not a formula, but a term in such definitions. This definition creates one new pattern and one new constructor with absolute name "`k3_real_1`". The translation of the typing is similar to that of the previous example:
"`forall([X,Y], implies( and(v1_arytm(X),v1_arytm(Y)), v1_arytm(k3_real_1(X,Y))))`".
As suggested by the Mizar keyword "`equals`", the *Definiens* is translated as equality:
"`equal(k3_real_1(X,Y), +(x,(-y)))`", more exactly:
"`forall([X,Y], implies( and(v1_arytm(X),v1_arytm(Y)), equal(k3_real_1(X,Y),k3_arytm(X,k1_real_1(Y)))))`"

This *Functor-Definition*, uses the single *Correctness-Condition* introduced by the keyword "`correctness`". This is mostly used as a shorthand, when proving the existence and uniqueness conditions is trivial and the system can prove them without any hint.

3.5 Structure Definitions

Types in Mizar are defined either as modes or as structures. Structures correspond (to some extent) to the "product" types of various other languages. The syntax of *Structure-Definition* is:

`struct [ (` *Ancestors* `) ]` *Structure-Symbol* `[ over` *Loci* `] (#` *Fields* `#)` `;`

Example: The simplest structure in Mizar is "`1-sorted`" defined in article `struct_0`:

```
struct 1-sorted(\# carrier -> set \#);''
```

It has only one field "`carrier`" of the type "`set`" and no *Ancestors*. Encapsulating the most general type into structure has little mathematical use of itself. So the main use of such structure is to state in an abstract way the

properties, which are common to all other structures containing a "carrier" field. Such structures will have "1-sorted" as an *Ancestor*.

Example:

```
struct(1-sorted) TopStruct (# carrier -> set,
                              topology -> Subset-Family of the carrier #);
```

This is definition of the structure TopStruct from the article about Topological Spaces "pre_topc". This is not a topological space yet, the common practice in Mizar in such cases is first to define the underlying structure (here the TopStruct) and after that the needed type (TopSpace in this case) is defined as the structure with some additional properties. Here, the definition of TopSpace is:

```
definition
  mode TopSpace is TopSpace-like TopStruct;
end;
```

where the attribute "TopSpace-like" (defined also in pre_topc) names the usual conditions for a Subset-Family (the "topology" slot of TopStruct) to be a topology.

Structures can define their ancestor structures. All *Fields* of an ancestor must also be *Fields* of the defined structure. In our example, there is given one ancestor "1-sorted".

Ancestors of structures are treated in a way similar to the treating of mother types of modes by the system, i.e. they also define the type hierarchy.

The *Structure-Symbol* is "TopStruct" in our example. The optional syntax "[over *Loci*]" is not used there. If used, it parameterizes the structure in the same way as modes can be parameterized, i.e. structures can also have any number of arguments. Finally, the structure defines two *Fields*:
"carrier" of the type "set", and
"topology" of the type "Subset-Family of the carrier".

We will now explain the effects of the *Structure-Definition* on the constructor level.

Any structure defines a new type constructor. Such constructors are also called "aggregate types", and the standard Mizar symbol for them is "l". So

the absolute name of the aggregate type in our example is "`l1_pre_topc`". Such types behave in exactly the same way as the mode types, i.e. variables can be reserved for them, *Adjective-Cluster*s can be added to them as a prefix, etc.

Ancestors become the mother types of the new aggregate type, so in our example, there is one mother type "`1-sorted`" (its absolute name is "`l1_struct_0`").

The *Fields* of a structure give rise to the so called "selector constructors". The standard Mizar symbol for selectors is "`u`". In our case, the structure contains two selectors, but the selector "`carrier`" is inherited from the ancestor "`1-sorted`" (`l1_struct_0`), where it was defined for the first time. So its absolute name is "`u1_struct_0`". The selector "`topology`" is defined for the first time in our example, so its absolute name is "`u1_pre_topc`".

The meaning of a selector is a function operating on the aggregate type. So e.g. "`carrier`" takes arguments of the type "`1-sorted`" (or any of its specializations, like `TopStruct`) and its result type is "`set`". The keyword "`the`" and "`of`" apply when using selectors, so e.g. if X is of type "`1-sorted`", the correct Mizar syntax is "`the carrier of X`" (instead of just e.g. "`carrier X`", which would be the case for normal Mizar functions).

The second selector "`topology`" takes argument (we denote it X here) of the type "`TopStruct`" and its result type is "`Subset-Family of the carrier of X`". In this result type, the selector "`carrier`" is already used in the parameter of the mode "`Subset-Family`". Such result types (parameterized by the "`carrier`" term) occur quite often for selectors.

Finally, we need something to create the desired structure, when the *Fields* are given. This is solved in Mizar by creating "aggregate functor" for the structure. The standard Mizar symbol for aggregate functors is "`g`", so for `TopStruct`, its absolute name is "`g1_pre_topc`". The symbol for the aggregate functor is in Mizar the same as the symbol for the aggregate type, i.e. `TopStruct`. Given properly typed arguments, e.g. by following global reservations:
"`reserve X for set;`"
"`reserve Y for Subset-Family of X;`"
the Mizar term "`TopStruct(#X,Y#)`" means applying the aggregate functor "`TopStruct`" ("`g1_pre_topc`") to arguments X and Y, yielding an object of

the aggregate type "TopStruct" ("l1_pre_topc").

Obviously, it holds then that
"X = the carrier of TopStruct(#X,Y#)" and
"Y = the topology of TopStruct(#X,Y#)".

Now we will explain, how *Structure-Definitions* are translated into first-order syntax.

Aggregate types are translated again as predicates, selectors and aggregate functors are translated as functions. First, we need to translate the type (ancestor) hierarchy. This is the same as for *Mode-Definition*s. So in our example, it is:
"TopStruct(X) implies 1-sorted(X)", in absolute notation:
"forall([X], implies(l1_pre_topc(X), l1_struct_0(X))).".

Next we need to state the typing theorems for the selectors. This is the same as typing theorems for normal Mizar functors. So for the selector "carrier" it was done when translating the definition of the structure "1-sorted":
"1-sorted(X) implies set(carrier(X))",
since "set" pruning applies, no typing theorem was actually created for "carrier".

For the selector "topology", we get:
"TopStruct(X) implies Subset-Family(topology(X),carrier(X))".
Since Subset-Family (defined in article "setfam_1") is expandable mode with quite long expansion, we do not translate it in the following absolute name translation of the previous formula:

"forall([X], implies(l1_pre_topc(X),
Subset-Family(u1_pre_topc(X),u1_struct_0(X)))".

(however, the reader is encouraged to try it as an exercise, using the definitions:
"mode Subset-Family of D is Subset of bool D;" and
"mode Subset of X is Element of bool X;",
where the absolute name of the functor "bool" (power set) is "k1_zfmisc_1").

Similarly, we need a typing theorem for the aggregate functor operating on the *Fields*:
"`(set(X) and Subset-Family(Y,X)) implies TopStruct(TopStruct(X,Y)).`"
This looks quite confusing without the absolute notation, so here it is (pruning the "`set`" and not translating "`Subset-Family`"):
"`forall([X,Y], implies( Subset-Family(Y,X),`
`l1_pre_topc(g1_pre_topc(X,Y))))`".

Finally, we need to encode the interdependence of these constructors. This is done by stating that the structures are freely generated by its *Fields*, using the aggregate functor:
"`TopStruct(X) implies X = TopStruct(carrier(X),topology(X));`",
in absolute notation:
"`forall([X], implies( l1_pre_topc(X),`
`equal(X,g1_pre_topc(u1_struct_0(X),u1_pre_topc(X)))))`".

3.6 Cluster Registrations

We have already mentioned, that *Cluster-Registrations* define for the system the special rules of attribute handling. They either state non emptiness of attributed types (*Existential-Registration*) or define attribute propagation (*Conditional-Registration*, *Functorial-Registration*).

3.6.1 Existential-Registration

The syntax of *Existential-Registration* is:
`cluster` *Adjective-Cluster Type-Expression* `;` *Correctness-Conditions.*

Existential-Registrations assert the existence (non emptiness of the extension) of a type with some *Adjective-Cluster* prefix.

Example:

```
definition
 cluster cardinal set;
  existence
   proof ...
   end;
end;
```

in article `card_1` asserts the existence of the type "`set`" with the *Adjective* "`cardinal`". Whenever a variable is being reserved for a type with some non empty *Adjective-Cluster*, the non emptiness of the type's extension is checked by the system.

We already noted how types with *Adjective-Cluster* are translated, so the translation is a simple existence formula in such cases:
"`exists([X], (cardinal(X) and set(X)))`".
In absolute notation and pruning the "`set`":
"`exists([X], v1_card_1(X))`".

3.6.2 Conditional-Registration

The syntax of *Conditional-Registration* is:
`cluster` *Adjective-Cluster* `->` *Adjective-Cluster Type-Expression* `;` *Correctness-Conditions* .

*Conditional-Registration*s give rules for attribute propagation.

Example:

```
definition
 cluster cardinal -> Ordinal-like set;
  coherence
   proof ...
   end;
end;
```

from article `card_1` states that to any "`set`" with *Adjective-Cluster* "`cardinal`" the system should add the *Adjective-Cluster* "`Ordinal-like`". A proof of this must be given after the keyword "`coherence`". *Conditional-Registration*s are easily translated as implication:
"`set(X) and cardinal(X) implies Ordinal-like(X)`".
In absolute syntax and after "`set`" pruning:
"`forall([X], implies( v1_card_1(X), v3_ordinal1(X)))`".

3.6.3 Functorial-Registration

The syntax of *Functorial-Registration* is:

`cluster` *Term-Expression* `->` *Adjective-Cluster* `;` *Correctness-Conditions*

Example:

```
definition let X be finite set;
 cluster Card X -> finite;
 coherence;
end;
```

from article `card_1` says that for any X with the type "`finite set`", the attribute "`finite`" applies also to the term "`Card X`" (the cardinality of X). While *Conditional-Registration*s define attribute propagation for some underlying type ("`set`" in the previous example), *Functorial-Registration*s do this for terms ("`Card X`" here). Their translation is also simple:
"`(finite(X) and set(x)) implies finite(Card(X))`".
In absolute syntax and after "`set`" pruning :
"`(forall([X], implies( v1_finset_1(X), v1_finset_1(k1_card_1(X)))))`".

4 Redefinitions

In Mizar, there can be predicate, attribute, type and functor redefinitions. Apart from the keyword "`redefine`", they have the same syntactic structure as the respective definitions. They can introduce new notation, change the definitional formula or the result types, or add new properties.

Some redefinitions do not touch the constructor level, they only define new notation.

Example:

```
definition let M,N;
 redefine pred M c= N;
 synonym M <=' N;
 pred M in N;
 synonym M <' N;
end;
```

are two redefinitions from article `card_1`. The variables here had previously been globally declared for the type Cardinal:

"`reserve K,M,N for Cardinal;`"

These redefinitions just create new patterns "`<=‘`" resp. "`<‘`" operating on `Cardinal`s. These patterns are from that place on equivalent to the old patterns "`c=`" (subset relation) resp. "`in`", and they are mapped to the same constructors as the old patterns.

If the constructor level is touched by the redefinition, Mizar usually creates a new constructor, but knows to some extent, that it is equivalent to its redefined counterpart. The treatment in Mizar is not complete, e.g. if several type redefinitions are possible, only the last defined is chosen, since Mizar implementation sometimes requires the terms to have unique result types.

This is no serious restriction for Mizar, since Mizar allows explicit typing (using the "`is`" keyword) of terms, and their "reconsidering", e.g.:

```
reserve A for finite Ordinal;
Lemma1: A is Integer
        proof ...
        end;
reconsider A as Integer by Lemma1;
Lemma2: A-A=0 by REAL_1:36;
```

Here, first A is reserved for the type "`finite Ordinal`", then it is proved, that A is also of the type "`Integer`". After that, we can retype ("`reconsider`") A as "`Integer`", and after that we can already apply the subtraction operation to A, since this subtraction is defined for reals and "`Integer`" (unlike "`Ordinal`") is a specialization of the type "`Real`".

Since we translate types (and attributes) as predicates, there is no trouble in having multiple "types" for a term. After the translation, these are simply several unit clauses holding about the term, e.g.:
"`finite(A) and Ordinal(A) and Integer(A) and Real(A)`".

So unlike Mizar, we do not create new constructors for redefinitions, we just translate the new facts stated by the redefinitions.

For attribute redefinitions, the only thing redefined can be the *Definiens*, e.g. for the previously given attribute "`limit`", the redefinition could be:

```
definition let M;
 redefine attr M is limit means
 for N <' M holds nextcard N <' M;
 compatibility
  proof ...
  end;
end;
```

After the keyword "`compatibility`", a proof must be given, that the new definition is equivalent to the old one. So the translation of such redefinition is simply a new equivalence formula:
"`limit(X) iff forall([Y], <‘(Y,X) implies <‘(nextcard(Y),X))`".

The situation is almost the same for predicates, but new *Predicate-Properties* can be given for them in the redefinition. They are translated exactly as in normal definitions.

For function and type redefinitions, there can additionally be a change in the result (mother) type of the redefined constructor. In such cases, we just add a new type hierarchy translation for the constructor, e.g.:

```
definition let x be Real;
  redefine func -x -> Real;
coherence by ARYTM:def 2;
```

is a redefinition of the previously mentioned definition of the unary function "`-`". The keyword "`coherence`" here starts a proof, that the new result type "`Real`" is correct (it is proved by reference to definition 2 in article `arytm`). The new typing theorem is:
"`Real(X) implies Real(k1_real_1(X))`".

5 Unimplemented Features of Mizar

Mizar *Scheme*s and Fraenkel terms are not currently translated, numbers are translated only in a very simple way.

Mizar *Scheme*s are used to express second-order assertions parameterized by functor or predicate variables. This makes it possible, to state principles like Separation or Induction in Mizar, e.g.:

```
scheme Separation { A()-> set, P[set] } :
   ex X st for x holds x in X iff x in A() & P[x]
    proof ...
    end;
```

from article `boole` states that for any set "`A()`" and any formula "`P`" with a free variable of the type "`set`", there is a subset of "`A()`" of elements with the property "`P`".

Similarly,

```
scheme Ind { P[Nat] } :
 for k holds P[k]
   provided
   P[0] and
   for k st P[k] holds P[k + 1];
```

from article `nat_1` is the usual induction principle for formulas with a free variable of type "`Nat`" (natural numbers).

Another feature of Mizar is the Fraenkel ("setof") operator, which (in accordance with the Separation principle) creates sets of elements that satisfy a given formula.

Example:

```
{i where i is Nat: i < 5}
```

is the set of natural numbers smaller than 5. The result type of such terms is the largest type "`set`".

Schemes and Fraenkel terms cannot be directly translated into first-order syntax. These features are relatively rare in MML and their usage is quite restricted (full instantiation of a scheme must be given before it can be applied), so a very large part of MML can be translated even if these features stay unhandled. We think the performance gap between current first order provers and higher order provers justifies such approach.

Theoretically, the Fraenkel terms and *Scheme*s could be completely removed using the standard set-theoretic elimination procedures (previous work with Otter in this direction is described in [Belinfante 96]), however, given

the advanced constructor system in Mizar, this could lead to very large (and thus practically unusable) expansions.

We believe, that the solution lies rather in modifying the current efficient first order provers to handle the "real mathematics" (at least to the extent the Mizar checker handles it today).

This also applies to the Mizar built-in handling of numbers. The current (most simple) translation creates for each mentioned number a new constant. However, Mizar can evaluate some simple arithmetic expressions like "`(5+3)*4`".

Finally, we should note, that our translation does not go into the Mizar proofs. Given a Mizar theorem, we simply collect the references that were used to prove it, and save them hidden behind a dfg comment in the resulting dfg file, for possible further generation of a first-order reproof task. Since there are currently more than 35.000 theorems in the MML, we believe this is sufficient for any serious experiments with reproving.

However, since current first-order provers have already defined simple proof formats, and proof checkers for such formats are available, translating Mizar proofs to such formats would have the additional benefit of the possibility of independent cross-checking of the Mizar checker by other first order checkers. Additionally, if reproof tasks were generated from the lemmas occurring in the Mizar proofs too, their number would rise to hundreds of thousands.

So since the main job (translating to first-order syntax) is already done, translating the proofs is probably worth the additional effort and will be done in the future.

6 Computer Implementation

Is done using the sources of the Mizar system (currently not public, implemented in Delphi, FPC and partially GPC and Kylix), mainly the part which inserts checked Mizar articles into common Mizar database (Exporter). The main tool (fo_tool) creates a first order database with structure similar to the normal Mizar database.

Initial support was written for Lisp and Prolog syntax (the Prolog syntax based on previous work of Czeslaw Bylinsky for the ILF group), the complete

export is into DFG syntax (as used e.g. by the SPASS prover). Some functions for experimenting with the translated database and user interface where implemented in Emacs Lisp (as part of author's Mizar mode), some initial experiments (mainly to check the correctness of the translation) were conducted with the SPASS prover.

7 Relation to the ILF Project

The probably largest effort at translating the MML to first order syntax was carried out several years ago by the ILF group [Dahn 97]. MML articles were exported into a rich Prolog format (using a special program made by Czeslaw Bylinsky), and this format was further processed by parts of the ILF system (written mostly in Prolog) with the main result being a block proof format [Dahn and Wolf 94] of MML articles.

Articles in block proof format were then inserted into a SQL database (Postgres), together with some other auxiliary data (e.g. bibliographical info, etc.). Export functions into several first order formats (TPTP, Otter, dfg, ...) were written for this database. TPTP problems generated in this way from several articles were included into the TPTP library [Suttner and Sutcliffe 98], problems in other formats are available on demand from the ILF server.

The relationship of the work described here to the ILF system is following: The ILF project (unfortunately) stopped some time ago, with the MML part of it not finished completely and no longer being updated. Finishing and updating it could still be quite time consuming process, owing to the size of the ILF codebase and the generic approach taken there (several levels of translation, using external SQL database).

Since the author's need was to have quickly a simple tool for complete and uniform first order translation of the MML, we rather reused parts of the Mizar codebase (Exporter) together with an updated version of the above mentioned Prolog export program written by C. Bylinsky. This made it possible to do the translation in a couple of days, with only minor (mainly output format) changes to the Mizar codebase. However, this approach is far less generic than the ILF approach, so the long term goal should probably be updating or completing of the Mizar-related parts of the ILF system.

8 Related and Future Work

An important part of the translation are the scripts that generate proving problems from the translated Mizar database. Such scripts have to mimic the Mizar theory inclusion mechanism closely, add special axioms for things that are obvious to the Mizar checker, and possibly try to apply some filtering to handle large theories. Due to space constraints, we do not describe them here, they will probably be described in another article about experiments with theorem provers on the translated database.

Experimenting with theorem provers (currently E and SPASS) is work in progress at the time of writing this article. The main purpose behind the translation is to train theorem provers on the large database, and possibly also implement discovery systems that could make use of it.

The most recent experiments show, that the "well-typing" inferences that have to be carried out by the provers are much less efficient than the type system implementation in Mizar. So another work which is currently under way, is to add an efficient Mizar-like type system to theorem provers. It seems that implementation of type discipline will be a necessary feature for theorem provers to be able to handle large theories. For example, the SPASS prover, which implements Mizar types to some extent, seems to perform much better in advanced theories, than the heavily optimized recent CASC winners.

As already noted, another feature, that will eventually have to be added to theorem provers to handle theories based on ZFC (i.e. most of current mathematics), is efficient implementation of infinite schemes of first-order axioms. In accordance with e.g. [**?**] we also believe, that theorem provers should be able to handle numbers efficiently.

We think a good way to boost the development of all these features for current theorem provers, would be to have e.g. a special part of CASC devoted to proving in large structured theory ensembles like MML. Detailed rules of such competition concerning e.g. allowed machine learning or type translation methods would have to be thought up.

References

[Belinfante 96] Johan Belinfante, On a Modification of Godel's Algorithm for Class Formation, Association for Automated Reasoning Newsletter,

No. 34, pp. 1015 (1996)]

[Bonarska 90] Bonarska, E., An Introduction to PC Mizar, Fondation Ph. le Hodey, Brussels, 1990.

[CARD_1] Grzegorz Bancerek, Cardinal numbers. Journal of Formalized Mathematics, 1, 1989.

[Dahn 97] First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. Ingo Dahn and Christoph Wernhard Proceedings of the International Workshop on First order Theorem Proving. RISC-Linz Report Series, No. 97-50, Johannes Kepler Universität Linz, 1997

[Dahn 98] Ingo Dahn. Interpretation of a Mizar-like Logic in First Order Logic. Proceedings of FTP 1998. pp. 137-151.

[Dahn and Wolf 94] B. I. Dahn and A. Wolf. Journal for Information Processing and Cybernetics (EIK), (5–6): pp. 261–276, 1994.

[Goguen 92] J.Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction formultiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science, 105(2):217-273,1992.

[Hahnle et all 96] R. Hahnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFGSchwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.

[McCune 94] McCune, W. W., OTTER 3.0 Reference Manual and Guide, Technical Report ANL-94/6, Argonne National Laboratory, Argonne, Illinois (1994).

[Muzalewski 93] Muzalewski, M., An Outline of PC Mizar, Fondation Philippe le Hodey, Brussels, 1993.

[PRE_TOPC] Beata Padlewska and Agata Darmochwal, Topological spaces and continuous functions. Journal of Formalized Mathematics, 1, 1989.

[REAL_1] Krzysztof Hryniewiecki, Basic properties of real numbers. Journal of Formalized Mathematics, 1, 1989.

[Rudnicky 92] Rudnicki, P., An Overview of the Mizar Project, Proceedings of the 1992 Workshop on Types for Proofs and Programs, Chalmers University of Technology, Bastad, 1992.

[Schumman 2001] J. M. Schumann, Automated Theorem-Proving in Software Engineering. Springer-Verlag, 2001.

[STRUCT_0] Mizar Mathematical Library Committee, Preliminaries to Structures. Journal of Formalized Mathematics, 1, 1989.

[SUBSET_1] Zinaida Trybulec, Properties of subsets. Journal of Formalized Mathematics, 1, 1989.

[Suttner and Sutcliffe 98] C. Suttner and G. Sutcliffe. The TPTP problem library (TPTP v2.2.0). Technical Report 9704, Department of Computer Science, James Cook University, Townsville, Australia, 1998.

[Syntax] Syntax of the Mizar Language available online at http://mizar.uwb.edu.pl/language/syntax.html

[Weidenbach et all 99] Weidenbach C., Afshordel B., Brahm U., Cohrs C., Engel T., Keen R., Theobalt C. and Topic D., System description: Spass version 1.0.0, in H. Ganzinger, ed., '16th International Conference on Automated Deduction, CADE-16', Vol. 1632 of LNAI, Springer, pp 314-318

[Wiedijk 99] Freek Wiedijk. Mizar: An impression. Unpublished paper, 1999. http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz.

A Syntax of the Mizar Language

As published at http://mizar.uwb.edu.pl/language/syntax.txt.

```
Last modified: June 26, 2000

   Article

Mizar-Article =
   "environ"
      Environment-Declaration
```

```
   "begin"
      Text-Proper .

   Environment

Environment-Declaration = { Directive } .

Directive =
   Vocabulary-Directive |
   Library-Directive |
   Requirement-Directive .

Vocabulary-Directive =
   "vocabulary" Vocabulary-Name { "," Vocabulary-Name } ";" .

Vocabulary-Name = File-Name .

Library-Directive =
   ( "notation" |
   "constructors" |
   "clusters" |
   "definitions" |
   "theorems" |
   "schemes" ) Article-Name { "," Article-Name } ";" .

Article-Name = File-Name .

Requirement-Directive = "requirements" Requirement { "," Requirement } ";" .

Requirement = File-Name .



   Text Proper

Text-Proper =
   { Text-Item }
   { Section } .
```

```
Section =
   "begin"
   { Text-Item } .

Text-Item =
   Reservation |
   Definitional-Item |
   Structure-Definition |
   Theorem |
   Scheme |
   Auxiliary-Item |
   Canceled-Theorem .

Reservation = "reserve" Reservation-Segment { "," Reservation-Segment
               } ";" .

Reservation-Segment = Reserved-Identifiers "for" Type-Expression .

Reserved-Identifiers = Identifier { "," Identifier } .

Definitional-Item = Definitional-Block ";" .

Definitional-Block =
   "definition"
      { Definition-Item | Definition }
   [ "redefine"
      { Definition-Item | Definition } ]
   "end" .

Definition-Item = Loci-Declaration | Permissive-Assumption |
                  Auxiliary-Item .

Loci-Declaration = "let" Qualified-Variables [ "such" Conditions ] ";" .

Permissive-Assumption = Assumption .

Definition =
```

```
   Structure-Definition |
   Mode-Definition |
   Functor-Definition |
   Predicate-Definition |
   Attribute-Definition |
   Canceled-Definition |
   Cluster-Registration .

Structure-Definition =
   "struct" [ "(" Ancestors ")" ] Structure-Symbol [ "over" Loci ]
      "(#" Fields "#)" ";" .

Ancestors = Type-Expression { "," Type-Expression } .

Structure-Symbol = Symbol .

Loci = Locus { "," Locus } .

Fields = Field-Segment { "," Field-Segment } .

Locus = Variable-Identifier .

Variable-Identifier = Identifier .

Field-Segment = Selector-Symbol { "," Selector-Symbol } Specification .

Selector-Symbol = Symbol .

Specification = "->" Type-Expression .

Mode-Definition =
   "mode" Mode-Pattern ( [ Specification ] [ "means" Definiens ] ";"
   Correctness-Conditions | "is" Type-Expression ";" )
   { Mode-Synonym } .

Mode-Pattern = Mode-Symbol [ "of" Loci ] .

Mode-Symbol = Symbol .
```

```
Mode-Synonym = "synonym" Mode-Pattern ";" .

Definiens = Simple-Definiens | Conditional-Definiens .

Simple-Definiens =
   [ ":" Label-Identifier ":" ] ( Sentence | Term-Expression ) .

Label-Identifier = Identifier .

Conditional-Definiens =
   [ ":" Label-Identifier ":" ] Partial-Definiens-List
      [ "otherwise" ( Sentence | Term-Expression ) ] .

Partial-Definiens-List = Partial-Definiens { "," Partial-Definiens } .

Partial-Definiens = ( Sentence | Term-Expression ) "if" Sentence .

Functor-Definition =
   "func" Functor-Pattern [ Specification ] [ ( "means" | "equals" )
   Definiens ] ";"
   Correctness-Conditions
   { Functor-Property }
   { Functor-Synonym } .

Functor-Pattern =
   [ Functor-Loci ] Functor-Symbol [ Functor-Loci ] |
   Left-Functor-Bracket Loci Right-Functor-Bracket .

Functor-Property = "commutativity" Justification ";" .

Functor-Synonym = "synonym" Functor-Pattern ";" .

Functor-Loci = Locus | "(" Loci ")" .

Functor-Symbol = Symbol .

Left-Functor-Bracket = Symbol .
```

```
Right-Functor-Bracket = Symbol .

Predicate-Definition =
   "pred" Predicate-Pattern [ "means" Definiens ] ";"
   Correctness-Conditions
   { Predicate-Property }
   { Predicate-Synonym } .

Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .

Predicate-Property =
   "symmetry" Justification ";" |
   "connectedness" Justification ";" |
   "reflexivity" Justification ";" |
   "irreflexivity" Justification ";" .

Predicate-Synonym = ( "synonym" | "antonym" ) Predicate-Pattern ";" .

Predicate-Symbol = Symbol .

Attribute-Definition =
   "attr" Attribute-Pattern "means" Definiens ";"
   [ Correctness-Condition ]
   { Attribute-Synonym } .

Attribute-Pattern = Locus "is" Attribute-Symbol .

Attribute-Synonym =
   ( "synonym" | "antonym" ) ( Attribute-Pattern | Predicate-Pattern ) ";" .

Attribute-Symbol = Symbol .

Canceled-Definition = "canceled" [ Numeral ] ";" .

Cluster-Registration =
   Existential-Registration |
   Conditional-Registration |
```

```
   Functorial-Registration .

Existential-Registration =
   "cluster" Adjective-Cluster Type-Expression ";"
   Correctness-Conditions .

Adjective-Cluster = { Adjective } .

Adjective = [ "non" ] Attribute-Symbol .

Conditional-Registration =
   "cluster" Adjective-Cluster "->" Adjective-Cluster Type-Expression ";"
   Correctness-Conditions .

Functorial-Registration =
   "cluster" Term-Expression "->" Adjective-Cluster ";"
   Correctness-Conditions .

Correctness-Conditions =
   { Correctness-Condition }
   [ "correctness" Justification ";" ] .

Correctness-Condition =
   "existence" Justification ";" |
   "uniqueness" Justification ";" |
   "coherence" Justification ";" |
   "compatibility" Justification ";" |
   "consistency" Justification ";" .

Theorem = "theorem" Compact-Statement .

Scheme =
   [ "scheme" ] Scheme-Identifier "{" Scheme-Parameters "}" ":"
      Scheme-Conclusion
   [ "provided"
      Scheme-Premise
      { "and" Scheme-Premise } ]
   Justification ";" .
```

```
Scheme-Identifier = Identifier .

Scheme-Parameters = Scheme-Segment { "," Scheme-Segment } .

Scheme-Conclusion = Sentence .

Scheme-Premise = Proposition .

Scheme-Segment = Predicate-Segment | Functor-Segment .

Predicate-Segment =
   Predicate-Identifier { "," Predicate-Identifier } "["
   Type-Expression-List "]" .

Predicate-Identifier = Identifier .

Functor-Segment =
   Functor-Identifier { "," Functor-Identifier } "(" [
   Type-Expression-List ] ")" Specification .

Functor-Identifier = Identifier .

Auxiliary-Item = [ "then" ] Statement | Private-Definition .

Canceled-Theorem = "canceled" [ Numeral ] ";" .

Private-Definition =
   Constant-Definition |
   Private-Functor-Definition |
   Private-Predicate-Definition .

Constant-Definition = "set" Equating-List ";" .

Equating-List = Equating { "," Equating } .

Equating = Variable-Identifier "=" Term-Expression .
```

```
Private-Functor-Definition =
   "deffunc" Private-Functor-Pattern "=" Term-Expression .

Private-Predicate-Definition =
   "defpred" Private-Predicate-Pattern "means" Sentence .

Private-Functor-Pattern = Functor-Identifier "(" [
   Type-Expression-List ] ")" .

Private-Predicate-Pattern =
   Predicate-Identifier "[" [ Type-Expression-List ] "]" .

Reasoning =
   { Reasoning-Item }
   [ "per" "cases" Simple-Justification ";"
   ( Case-List | Suppose-List ) ] .

Case-List = Case { Case } .

Case =
   "case" ( Proposition | Conditions ) ";"
   { Reasoning-Item } .

Suppose-List = Suppose { Suppose } .

Suppose =
   "suppose" ( Proposition | Conditions ) ";"
   { Reasoning-Item } .

Reasoning-Item = Auxiliary-Item | Skeleton-Item .

Skeleton-Item = Generalization | Assumption | Conclusion |
                Exemplification .

Generalization = "let" Qualified-Variables [ "such" Conditions ] ";" .

Assumption =
   Single-Assumption | Collective-Assumption | Existential-Assumption .
```

```
Single-Assumption = "assume" Proposition ";" .

Collective-Assumption = "assume" Conditions ";" .

Existential-Assumption =
   "given" Qualified-Variables [ "such" Conditions ] ";" .

Conclusion = ( "thus" | "hence" ) Statement .

Exemplification = "take" Example { "," Example } ";" .

Example = Term-Expression | Variable-Identifier "=" Term-Expression .

Statement = [ "then" ] Linkable-Statement | Diffuse-Statement .

Linkable-Statement =
   Compact-Statement |
   Choice-Statement |
   Type-Changing-Statement |
   Iterative-Equality .

Compact-Statement = Proposition Justification ";" .

Choice-Statement =
   "consider" Qualified-Variables [ "such" Conditions ]
   Simple-Justification ";" .

Type-Changing-Statement =
   "reconsider" Type-Change-List "as" Type-Expression
   Simple-Justification ";" .

Type-Change-List =
   ( Equating | Variable-Identifier ) { "," ( Equating |
   Variable-Identifier ) } .

Iterative-Equality =
   Term-Expression "=" Term-Expression Simple-Justification
```

```
                        ".=" Term-Expression Simple-Justification
                     { ".=" Term-Expression Simple-Justification } ";" .

Diffuse-Statement =
   [ Label-Identifier ":" ]
   "now"
      Reasoning
   "end" ";" .

Justification = Simple-Justification | Proof .

Simple-Justification = Straightforward-Justification | Scheme-Justification .

Proof =
   ( "proof" | "@proof" )
      Reasoning
   "end" .

Straightforward-Justification = [ "by" References ] .

Scheme-Justification = "from" Scheme-Identifier [ "(" References ")" ] .

References = Reference { "," Reference } .

Reference = Local-Reference | Library-Reference .

Local-Reference = Label-Identifier .

Library-Reference =
   Article-Name ":" ( Theorem-Number | "def" Definition-Number )
      { "," ( Theorem-Number | "def" Definition-Number ) } .

Theorem-Number = Numeral .

Definition-Number = Numeral .

Conditions = "that" Proposition { "and" Proposition } .
```

```
Proposition = [ Label-Identifier ":" ] Sentence .

Sentence = Formula-Expression .
```

Expressions

```
Formula-Expression =
   "(" Formula-Expression ")" |
   Atomic-Formula-Expression |
   Quantified-Formula-Expression |
   Formula-Expression "&" Formula-Expression |
   Formula-Expression "or" Formula-Expression |
   Formula-Expression "implies" Formula-Expression |
   Formula-Expression "iff" Formula-Expression |
   "not" Formula-Expression |
   "contradiction" |
   "thesis" .

Atomic-Formula-Expression =
   [ Term-Expression-List ] Predicate-Symbol [ Term-Expression-List ] |
   Predicate-Identifier [ "[" Term-Expression-List "]" ] |
   Term-Expression "is" { Adjective } |
   Term-Expression "is" Type-Expression .

Quantified-Formula-Expression =
   "for" Qualified-Variables [ "st" Formula-Expression ]
      ( "holds" Formula-Expression | Quantified-Formula-Expression ) |
   "ex" Qualified-Variables "st" Formula-Expression .

Qualified-Variables =
   Implicitly-Qualified-Variables |
   Explicitly-Qualified-Variables |
   Explicitly-Qualified-Variables "," Implicitly-Qualified-Variables .

Implicitly-Qualified-Variables = Variables .
```

```
Explicitly-Qualified-Variables = Qualified-Segment { "," Qualified-Segment } .

Qualified-Segment = Variables Qualification .

Variables = Variable-Identifier { "," Variable-Identifier } .

Qualification = ( "being" | "be" ) Type-Expression .

Type-Expression = "(" Type-Expression ")" | Adjective-Cluster Radix-Type .

Radix-Type =
   Mode-Symbol [ "of" Term-Expression-List ] |
   Structure-Symbol [ "over" Term-Expression-List ] .

Type-Expression-List = Type-Expression { "," Type-Expression } .

Term-Expression =
   "(" Term-Expression ")" |
   [ Arguments ] Functor-Symbol [ Arguments ] |
   Left-Functor-Bracket Term-Expression-List Right-Functor-Bracket |
   Functor-Identifier "(" [ Term-Expression-List ] ")" |
   Structure-Symbol "(#" Term-Expression-List "#)" |
   Variable-Identifier |
   "{" Term-Expression [ Postqualification ] ":" Sentence "}" |
   Numeral |
   Term-Expression "qua" Type-Expression |
   "the" Selector-Symbol "of" Term-Expression |
   "the" Selector-Symbol |
   Private-Definition-Parameter |
   "it" .

Arguments = Term-Expression | "(" Term-Expression-List ")" .

Term-Expression-List = Term-Expression { "," Term-Expression } .

Postqualification =
   "where" Postqualifying-Segment { "," Postqualifying-Segment } .
```

```
Postqualifying-Segment =
   Postqualified-Variable { "," Postqualified-Variable } "is"
   Type-Expression .

Postqualified-Variable = Identifier .

Private-Definition-Parameter =
   "$1" | "$2" | "$3" | "$4" | "$5" | "$6" | "$7" | "$8" .
```

Appendix 1 - Example of the translation and reproving methods

In this appendix we show the complete process of translating and reproving a selected Mizar theorem. All the generated files mentioned here are also present in the *example* directory on the enclosed CD, the corresponding Mizar files are in the *mizar* directory there.

Our example is the theorem 42 in article CARD_1 (CARD_1:42) - injectivity of the function *alef*:

```
reserve A,B for Ordinal;
theorem :: CARD_1:42
  alef A = alef B implies A = B;
```

The function *alef* is in MML defined as follows (*T-Sequence* stands for transfinite sequence here, i.e. a function whose domain is an Ordinal):

```
reserve A,B for Ordinal;
reserve L,L1 for T-Sequence;
definition let A;
 func alef A -> set means
:: CARD_1:def 8
   ex L st it = last L & dom L = succ A & L.{} = Card NAT &
        (for B,y st succ B in succ A & y = L.B holds
                    L.succ B = nextcard union { y }) &
         for B,L1 st B in succ A & B <> {} & B is_limit_ordinal & L1 = L|B
                holds L.B = Card sup L1;
end;
```

Mizar proof of CARD_1:42 uses MML theorems ORDINAL1:24 and CARD_1:41 (the predicate <‘ is the strict ordering of ordinals, and in Mizar it is a synonym for the perdicate in).

```
reserve A,B for Ordinal;
theorem :: ORDINAL1:24
  A in B or A = B or B in A;

theorem :: CARD_1:41
  A in B iff alef A <‘ alef B;
```

The Mizar proof is as follows:

```
theorem
 Th42: alef A = alef B implies A = B
  proof assume
A1:  alef A = alef B;
A2:  now assume A in B; then
       alef A <‘ alef B by Th41;
      hence contradiction by A1;
     end;
     now assume B in A; then
       alef B <‘ alef A by Th41;
      hence contradiction by A1;
     end;
   hence thesis by A2,ORDINAL1:24;
  end;
```

The proof uses two lemmas, introduced by the `now assume A in B;` and `now assume B in A;` reasoning items. Both lemmas just show, that their assumptions lead to contradiction, i.e. neither of them can be true, and hence we can prove the thesis that `A = B` using the disjunction ORDINAL1:24. Note that in both lemmas, the contradiction in Mizar is inferred from the fact, that `alef A = alef B` and `alef B <‘ alef A` (or its opposite) cannot be simultaneously true. This rule is built-in in Mizar for a more general case, i.e. the predicate `in` (for which `<‘` is just a synonym) is known to be irreflexive to the Mizar checker.

Following is the reproval problem of CARD_1:42 generated for SPASS with the default signature filtering. For better understanding, its back-translation from the constructor format to the Mizar user symbols is shown next to it.

```
begin_problem(t42_card_1).
list_of_descriptions.
name({*t42_card_1*}).
author({*Mizar Mathematical Library*}).
status(unsatisfiable).
description({*Problem generated from MML by MPTP*}).
end_of_list.
```

```
list_of_symbols.
functions[

% Article functors:
(k3_card_1,1),

% Numerals' arities:
% Constants' arities:
% Scheme functors:
(setof,0)
].

predicates[
% Scheme predicates:
% Article predicates:
(v1_card_1,1),  (v3_ordinal1,1),
(r2_hidden,2),  (v2_ordinal1,1),
(v1_ordinal1,1)
].
end_of_list.
list_of_formulae(axioms).

% Constructor types:
% Mode existence:
% Constructor properties:
formula(
forall([B1,B2],
  not(
    and( r2_hidden(B1,B2),
      r2_hidden(B2,B1)))),
p8_r2_hidden).

% Existential clusters:
% Functor clusters:
formula(forall([A1],
implies(
  v3_ordinal1(A1),
  and( v1_ordinal1(k3_card_1(A1)),
and( v2_ordinal1(k3_card_1(A1)),
and( v3_ordinal1(k3_card_1(A1)),
v1_card_1(k3_card_1(A1)))))),
fc0_card_1).
```

```
list_of_symbols.
functions[

% Article functors:
(alef,1),

% Numerals' arities:
% Constants' arities:
% Scheme functors:
(setof,0)
].

predicates[
% Scheme predicates:
% Article predicates:
(cardinal,1),   (ordinal,1),
(in,2), (epsilon-connected,1),
(epsilon-transitive,1)
].
end_of_list.
list_of_formulae(axioms).

% Constructor types:
% Mode existence:
% Constructor properties:
formula(
forall([B1,B2],
  not(
    and( in(B1,B2),
      in(B2,B1)))),
p8_r2_hidden).

% Existential clusters:
% Functor clusters:
formula(forall([A1],
implies(
  ordinal(A1),
  and( epsilon-transitive(alef(A1)),
      and( epsilon-connected(alef(A1)),
and( ordinal(alef(A1)),
     cardinal(alef(A1)))))),
fc0_card_1).
```

```
% Conditional clusters:
formula(forall([A1],
implies(
  v3_ordinal1(A1),
  and( v1_ordinal1(A1),
       v2_ordinal1(A1)))),
cc0_ordinal1).

formula(forall([A1],
implies(
  v1_card_1(A1),
  and( v1_ordinal1(A1),
and( v2_ordinal1(A1),
     v3_ordinal1(A1))))),
cc0_card_1).

formula(forall([A1],
implies(
  and( v1_ordinal1(A1),
       v2_ordinal1(A1)),
  v3_ordinal1(A1))),
cc1_ordinal1).

% Requirements:
% Special non-DB formulas:
% Direct references:

formula(
forall([B1,B2],
implies(
  and( v3_ordinal1(B1),
       v3_ordinal1(B2)),
  and(
  not(
    and( r2_hidden(B1,B2),
      not( r2_hidden(k3_card_1(B1),
,           k3_card_1(B2))))),
    not(
      and( r2_hidden(k3_card_1(B1),
           k3_card_1(B2)),
        not( r2_hidden(B1,B2)))))),
t41_card_1).
```

```
% Conditional clusters:
formula(forall([A1],
implies(
  ordinal(A1),
  and( epsilon-transitive(A1),
       epsilon-connected(A1)))),
cc0_ordinal1).

formula(forall([A1],
implies(
  cardinal(A1),
  and( epsilon-transitive(A1),
and( epsilon-connected(A1),
     ordinal(A1))))),
cc0_card_1).

formula(forall([A1],
implies(
  and( epsilon-transitive(A1),
       epsilon-connected(A1)),
  ordinal(A1))),
cc1_ordinal1).

% Requirements:
% Special non-DB formulas:
% Direct references:

formula(
forall([B1,B2],
implies(
  and( ordinal(B1),
       ordinal(B2)),
  and(
  not(
    and( in(B1,B2),
      not( in(alef(B1),
              alef(B2))))),
    not(
      and( in(alef(B1),
              alef(B2)),
       not( in(B1,B2)))))),
t41_card_1).
```

```
formula(
forall([B1,B2],
implies(
  and( v3_ordinal1(B1),
       v3_ordinal1(B2)),
  not(
    and(
    not( r2_hidden(B1,B2))
    ,and(
    not( equal(B1,B2)),
      not( r2_hidden(B2,B1)))))),
t24_ordinal1).

end_of_list.

list_of_formulae(conjectures).

formula(
forall([B1,B2],
implies(
  and( v3_ordinal1(B1),
       v3_ordinal1(B2)),
  not(
    and( equal(k3_card_1(B1),
         k3_card_1(B2)),
      not( equal(B1,B2))))),
t42_card_1).

end_of_list.

end_problem.
```

```
formula(
forall([B1,B2],
implies(
  and( ordinal(B1),
       ordinal(B2)),
  not(
    and(
    not( in(B1,B2))
    ,and(
    not( equal(B1,B2)),
      not( in(B2,B1)))))),
t24_ordinal1).

end_of_list.

list_of_formulae(conjectures).

formula(
forall([B1,B2],
implies(
  and( ordinal(B1),
       ordinal(B2)),
  not(
    and( equal(alef(B1),
               alef(B2)),
      not( equal(B1,B2))))),
t42_card_1).

end_of_list.

end_problem.
```

Note that in addition to the direct references `t24_ordinal1, t41_card_1` and the conjecture `t42_card_1` the signature filtering had allowed only five additional background formulas:

- `p8_r2_hidden` - asymmetry of the predicate `in`
- `fc0_card_1` - functor cluster adding attribute `cardinal` to the term `alef(A)`
- `cc0_card_1, cc0_ordinal1, cc1_ordinal1` - conditional clusters relating the attributes `cardinal, epsilon-transitive, epsilon-connected, ordinal`.

This problem was very easy for the SPASS prover, following is the proof found by SPASS, back-transformed for better readability to the user symbols. The statistics says, that only 62 clauses were derived in the standard given-clause theorem proving loop used by SPASS, until the proof was found.

```
SPASS V 2.1
SPASS beiseite: Proof found.
Problem: /home/urbanj/shared/MPTP/PROBLEMS/card_1/t42_card_1.dfg
SPASS derived 62 clauses, backtracked 0 clauses and kept 54 clauses.
SPASS allocated 616 KBytes.
SPASS spent     0:00:00.05 on the problem.
                0:00:00.02 for the input.
                0:00:00.01 for the FLOTTER CNF translation.
                0:00:00.00 for inferences.
                0:00:00.00 for the backtracking.
                0:00:00.01 for the reduction.


Here is a proof with depth 4, length 26 :
1[0:Inp] ||  -> ordinal(skc3)*.
2[0:Inp] ||  -> ordinal(skc2)*.
3[0:Inp] || equal(skc3,skc2)** -> .
7[0:Inp] ordinal(U) ||  -> epsilon-connected(U)*.
8[0:Inp] ordinal(U) ||  -> epsilon-transitive(U)*.
9[0:Inp] ||  -> equal(alef(skc3),alef(skc2))**.
15[0:Inp] || in(U,V)*+ in(V,U)* -> .
16[0:Inp] ordinal(U) ordinal(V) ||  -> in(U,V)* equal(U,V) in(V,U)*.
17[0:Inp] ordinal(U) ordinal(V) || in(U,V) -> in(alef(U),alef(V))*.
```

```
20[0:Res:2.0,17.0] ordinal(U) || in(U,skc2) -> in(alef(U),alef(skc2))*.
26[0:Res:2.0,7.0] ||  -> epsilon-connected(skc2)*.
27[0:Res:2.0,8.0] ||  -> epsilon-transitive(skc2)*.
29[0:Res:2.0,17.1] ordinal(U) || in(skc2,U) -> in(alef(skc2),alef(U))*.
38[0:Res:1.0,7.0] ||  -> epsilon-connected(skc3)*.
39[0:Res:1.0,8.0] ||  -> epsilon-transitive(skc3)*.
84[0:SpR:9.0,29.2] ordinal(skc3) || in(skc2,skc3)
                                    -> in(alef(skc2),alef(skc2))*.
86[0:SSi:84.0,1.0,38.0,39.0] || in(skc2,skc3)
                                     -> in(alef(skc2),alef(skc2))*.
87[0:Res:86.1,15.0] || in(skc2,skc3) in(alef(skc2),alef(skc2))* -> .
88[0:MRR:87.1,86.1] || in(skc2,skc3)* -> .
117[0:SpR:9.0,20.2] ordinal(skc3) || in(skc3,skc2)
                                      -> in(alef(skc2),alef(skc2))*.
120[0:SSi:117.0,1.0,38.0,39.0] || in(skc3,skc2)
                                      -> in(alef(skc2),alef(skc2))*.
121[0:Res:120.1,15.0] || in(skc3,skc2) in(alef(skc2),alef(skc2))* -> .
123[0:MRR:121.1,120.1] || in(skc3,skc2)* -> .
125[0:Res:16.2,123.0] ordinal(skc3) ordinal(skc2) ||
                               -> equal(skc3,skc2) in(skc2,skc3)*.
127[0:SSi:125.1,125.0,2.0,26.0,27.0,1.0,38.0,39.0]
     || -> equal(skc3,skc2) in(skc2,skc3)*.
128[0:MRR:127.0,127.1,3.0,88.0] ||  -> .
Formulae used in the proof : t42_card_1 cc0_ordinal1 p8_r2_hidden
                             t24_ordinal1 t41_card_1
```

It is natural that the SPASS proof is less human-readable than the Mizar proof, we will try to explain it a bit. SPASS tries to prove the conjecture from the supplied axioms by negating it, skolemizing and clausifying such input set of formulas, and looking for the contradiction in the resulting set of input clauses. The proof starts with input clauses (annotated by the [Inp] token) used in the proof (clauses 1,2,3,7,8,9,15,16,17) and ends by printing the formulas from which these clauses were generated (formulas `t42_card_1`, `cc0_ordinal1`, `p8_r2_hidden`, `t24_ordinal1`, `t41_card_1`). The clauses are printed in a *sorted sequent format*, which generally has the following form:

$$(1) \qquad S_i \ \texttt{||} \ A_j \ \texttt{->} \ C_k$$

In this format, S_i and A_j are all negative literals of the clause, and C_k are all its positive literals. SPASS employs special efficient sort inference mechanism

for suitable monadic predicates, which is the reason for distinguishing the sort literals S_i in the clauses from other negative literals A_j. However the semantics of (1) is the same as of:

$$(2) \qquad \texttt{||}\ S_i\ \texttt{,}\ A_j\ \texttt{->}\ C_k$$

which is just

$$\neg S_1 \vee ... \vee \neg S_{n_S} \vee \neg A_1 \vee ... \vee \neg A_{n_A} \vee C_1 \vee ... \vee C_{n_C}$$

The constants `skc2, skc3` are the skolem constants created during skolemization of the negated conjecture t42_card_1. Before the clausification, it would look this way, and it produces the input clauses 1, 2, 3 and 9:

```
and(
  and( ordinal(skc3),
       ordinal(skc2)),
  and( equal(alef(skc3),alef(skc2)),
       not( equal(skc3,skc2))))
```

This SPASS proof in fact proceeds similarly to the Mizar proof. It derives the lemmas

```
88 || in(skc2,skc3)* -> .
123 || in(skc3,skc2)* -> .
```

which together with the assumption

```
3[0:Inp] || equal(skc3,skc2)** -> .
```

and the translated theorem ORDINAL1:24

```
16[0:Inp] ordinal(U) ordinal(V) ||  -> in(U,V)* equal(U,V) in(V,U)*.
```

yield the contradiction. The lemmas 88 and 123 are derived using the translated theorem CARD_1:41 (clause 17), the asymmetry of `in` (clause 15) and also the assumption 3. The conditional cluster cc0_ordinal1 is actually a bit redundant part of the SPASS proof and its usage is reported only because it was used for the sort inferences yielding the clauses 120 and 127, which could be replaced by standard resolution.

We also show the SPASS proof for the corresponding unfiltered problem here. The full input is not shown here because of its length (178 formulas), it is however available on the enclosed CD (also with the version backtransformed to the user symbols).

```
SPASS V 2.1
SPASS beiseite: Proof found.
Problem: /home/urbanj/MPTP/PROBLEMS/card_1/t42_card_1.dfg
SPASS derived 610 clauses, backtracked 64 clauses and kept 578 clauses.
SPASS allocated 959 KBytes.
SPASS spent     0:00:00.40 on the problem.
                0:00:00.03 for the input.
                0:00:00.06 for the FLOTTER CNF translation.
                0:00:00.02 for inferences.
                0:00:00.00 for the backtracking.
                0:00:00.12 for the reduction.


Here is a proof with depth 3, length 27 :
54[0:Inp] ||  -> ordinal(skc15)*.
55[0:Inp] ||  -> ordinal(skc14)*.
101[0:Inp] || equal(skc15,skc14)** -> .
131[0:Inp] ||  -> equal(alef(skc15),alef(skc14))**.
201[0:Inp] || in(U,V)*+ in(V,U)* -> .
262[0:Inp] ordinal(U) ordinal(V) ||  -> in(U,V)* equal(U,V) in(V,U)*.
263[0:Inp] ordinal(U) ordinal(V) || in(U,V) -> in(alef(U),alef(V))*.
264[0:Inp] ordinal(U) ordinal(V) || in(alef(U),alef(V))* -> in(U,V).
323[0:Res:54.0,263.0] ordinal(U) || in(U,skc15)
                                    -> in(alef(U),alef(skc15))*.
344[0:Res:54.0,263.1] ordinal(U) || in(skc15,U)
                                    -> in(alef(skc15),alef(U))*.
345[0:Res:54.0,264.1] ordinal(U) || in(alef(skc15),alef(U))*
                                                -> in(skc15,U).
350[0:Res:262.3,101.0] ordinal(skc14) ordinal(skc15) ||
                            -> in(skc14,skc15) in(skc15,skc14)*.
358[0:Rew:131.0,323.2] ordinal(U) || in(U,skc15)
                                     -> in(alef(U),alef(skc14))*.
360[0:Rew:131.0,344.2] ordinal(U) || in(skc15,U)
                                     -> in(alef(skc14),alef(U))*.
361[0:Rew:131.0,345.1] ordinal(U) || in(alef(skc14),alef(U))*
```

```
                                                    -> in(skc15,U).
362[0:MRR:350.0,350.1,55.0,54.0] ||  -> in(skc15,skc14)* in(skc14,skc15).
363[0:Res:55.0,361.0] || in(alef(skc14),alef(skc14))* -> in(skc15,skc14).
364[0:Res:55.0,360.0] || in(skc15,skc14) -> in(alef(skc14),alef(skc14))*.
366[0:Res:55.0,358.0] || in(skc14,skc15) -> in(alef(skc14),alef(skc14))*.
437[1:Spt:362.0] ||  -> in(skc15,skc14)*.
438[1:MRR:364.0,437.0] ||  -> in(alef(skc14),alef(skc14))*.
936[1:Res:438.0,201.0] || in(alef(skc14),alef(skc14))* -> .
941[1:MRR:936.0,438.0] ||  -> .
942[1:Spt:941.0,362.0,437.0] || in(skc15,skc14)* -> .
943[1:Spt:941.0,362.1] ||  -> in(skc14,skc15)*.
944[1:MRR:363.1,942.0] || in(alef(skc14),alef(skc14))* -> .
945[1:MRR:366.0,366.1,943.0,944.0] ||  -> .
Formulae used in the proof : t42_card_1 p8_r2_hidden t24_ordinal1
                             t41_card_1
```

This proof is very similar to the previous one. It differs in the number of clauses derived and in not using the conditional cluster `cc0_ordinal1`. Because of the much higher number of input formulas, the search space has been much larger - 610 clauses were derived before the proof was found. This probably also caused that the order of inferencing was a bit different, and instead of the sort inference using `cc0_ordinal1`, standard resolution was used. Another minor difference is that the disjunction 362 was derived early, and used for the splitting rule implemented in SPASS. The clauses 437 and 942 are the cases introduced by this splitting, and for each of them contradiction is derived (clauses 941 and 945).

Finally we show the data generated in the fully automated proof attempted by the combined Mizar Proof Advisor/ SPASS architecture. The 30 hints (in the order of their expected importance) suggested for CARD_1:42 by the Mizar Proof Advisor trained on the previous MML proofs were following:

```
t39_card_1,t40_card_1,t41_card_1,t27_ordinal2,t41_ordinal1,
t24_ordinal1,t34_ordinal1,t35_ordinal2,t21_ordinal1,t19_ordinal1,
d2_ordinal1,d8_xboole_0,t32_card_1,t31_zfmisc_1,d5_card_1,
t27_card_1,d5_funct_1,t23_ordinal1,t52_ordinal2,t44_ordinal2,
t55_ordinal2,t56_ordinal2,t10_ordinal1,t45_ordinal2,t2_xboole_1,
t26_ordinal1,t22_ordinal1,t27_ordinal3,t33_ordinal1,t7_ordinal1
```

To these hints (and t42_card_1) background formulas were added, and the default signature filtering was applied, yielding 92 input formulas. This file is also too long to be included here, so readers are again advised to browse it on the enclosed CD. Note that in this case, all the references necessary for the Mizar proof are present among the first six hints belonging to only three Mizar articles, so if we stopped at that number of hints, both the number of direct references and the number of background formulas would be significantly lower.

Following is the proof of CARD_1:42 found by SPASS using the hints suggested by the Mizar Proof Advisor:

```
SPASS V 2.1
SPASS beiseite: Proof found.
Problem: /home/urbanj/MPTP/PROBLEMS/card_1/t42_card_1.dfg
SPASS derived 9289 clauses, backtracked 1260 clauses and kept 3396 clauses.
SPASS allocated 4289 KBytes.
SPASS spent     0:00:07.22 on the problem.
                0:00:00.05 for the input.
                0:00:00.10 for the FLOTTER CNF translation.
                0:00:00.37 for inferences.
                0:00:00.17 for the backtracking.
                0:00:04.58 for the reduction.


Here is a proof with depth 3, length 29 :
15[0:Inp] ||  -> ordinal(skc3)*.
16[0:Inp] ||  -> ordinal(skc2)*.
31[0:Inp] ||  -> in(U,succ(U))*.
43[0:Inp] || equal(skc3,skc2)** -> .
57[0:Inp] ||  -> equal(alef(skc3),alef(skc2))**.
106[0:Inp] || in(U,V)* c=(V,U) -> .
118[0:Inp] ordinal(U) || in(V,U)* -> ordinal(V).
159[0:Inp] ordinal(U) ordinal(V) || in(V,U) -> c=(succ(V),U)*.
168[0:Inp] ordinal(U) ordinal(V) ||  -> in(U,V)* equal(U,V) in(V,U)*.
169[0:Inp] ordinal(U) ordinal(V) || in(V,U) -> in(alef(V),alef(U))*.
170[0:Inp] ordinal(U) ordinal(V) || in(alef(U),alef(V))* -> in(U,V).
205[0:MRR:159.1,118.2] ordinal(U) || in(V,U) -> c=(succ(V),U)*.
206[0:MRR:169.1,118.2] ordinal(U) || in(V,U) -> in(alef(V),alef(U))*.
226[0:Res:16.0,206.0] || in(U,skc2) -> in(alef(U),alef(skc2))*.
229[0:Res:16.0,205.0] || in(U,skc2) -> c=(succ(U),skc2)*.
```

```
276[0:Res:16.0,170.1] ordinal(U) || in(alef(skc2),alef(U))*
                                                    -> in(skc2,U).
300[0:Res:15.0,206.0] || in(U,skc3) -> in(alef(U),alef(skc3))*.
360[0:Res:168.3,43.0] ordinal(skc2) ordinal(skc3) ||
                                     -> in(skc2,skc3) in(skc3,skc2)*.
389[0:Rew:57.0,300.1] || in(U,skc3) -> in(alef(U),alef(skc2))*.
397[0:MRR:360.0,360.1,16.0,15.0] ||  -> in(skc3,skc2)* in(skc2,skc3).
416[0:Res:16.0,276.0] || in(alef(skc2),alef(skc2))* -> in(skc2,skc2).
1545[0:Res:31.0,106.0] || c=(succ(U),U)* -> .
6766[0:Res:229.1,1545.0] || in(skc2,skc2)* -> .
7813[0:MRR:416.1,6766.0] || in(alef(skc2),alef(skc2))* -> .
10409[0:Res:389.1,7813.0] || in(skc2,skc3)* -> .
10759[0:SpR:57.0,226.1] || in(skc3,skc2) -> in(alef(skc2),alef(skc2))*.
10780[0:MRR:397.1,10409.0] ||  -> in(skc3,skc2)*.
10786[0:MRR:10759.1,7813.0] || in(skc3,skc2)* -> .
10787[0:MRR:10786.0,10780.0] ||  -> .
Formulae used in the proof : t42_card_1 t10_ordinal1 t7_ordinal1
                t23_ordinal1 t33_ordinal1 t24_ordinal1 t41_card_1
```

This problem was significantly harder for SPASS than the previous two reproval problems, 9289 clauses were generated before the proof was found. The proof is also a bit different from the previous two proofs, and though it is of little interest in this particular case, our combined architecture has thus really automatically produced a new proof in a nontrivial mathematical domain. From this point of view we can see that giving SPASS thirty hints instead of just the first six increases the chance of finding a new proof, which is probably going to be "less obvious" or perhaps sometimes "more innovative" than the most apparent solution coming from the strongest hints supplied by the bayesian memory of the Mizar Proof Advisor.

Appendix 2 - Structure of the database of results for MPTP

```
USE mptpresults;

/* Proposed structure of the database of results for MPTP */


/* Problem info independent of prover runs */

CREATE TABLE probleminfo (

  problem               VARCHAR(255),          /* Index on initial 20 chars */
  article               VARCHAR(8),            /* Article string */
  theorem_id            SMALLINT UNSIGNED,     /* Number in article */

  mizar_proof_length    INT UNSIGNED,
  direct_references_nr  INT UNSIGNED,          /* Without bg theory */
  direct_references     BLOB,                  /* List of their names */
  bg_references_nr      INT UNSIGNED,          /* bg theory */
  bg_references         BLOB,                  /* List of their names */
  all_references_nr     INT UNSIGNED,          /* With bg theory */

  conjecture_syms_nr    INT UNSIGNED,
  conjecture_syms       BLOB,
  direct_refs_syms_nr   INT UNSIGNED,
  direct_refs_syms      BLOB,
  problem_syms_nr       INT UNSIGNED,
  problem_syms          BLOB,                  /* All symbols */

  INDEX  xproblem               (problem(20)),
  INDEX  xarticle               (article),
  INDEX  xtheorem_id            (theorem_id),
  INDEX  xmizar_proof_length    (mizar_proof_length),
  INDEX  xdirect_references_nr  (direct_references_nr),
  INDEX  xbg_references_nr      (bg_references_nr),
  INDEX  xall_references_nr     (all_references_nr),
  INDEX  xdirect_refs_syms_nr   (direct_refs_syms_nr),
  INDEX  xproblem_syms_nr       (problem_syms_nr)
```

```
);


/* Additional problem info for proved tasks */

CREATE TABLE proved (

/* Primary problem identification */
  id                    INT NOT NULL AUTO_INCREMENT,
  problem               VARCHAR(255),           /* Index on initial 20 chars */
  article               VARCHAR(8),             /* Article string */
  theorem_id            SMALLINT UNSIGNED,      /* Number in article */
  format                ENUM('DFG','TPTP',
                             'LOP','OTTER'),
  prover                VARCHAR(255),           /* Name and version */

/* Result info */

  result                ENUM('PROOF','COMPLETION',
                             'TIMELIMIT','MEMLIMIT',
                             'KILLED','CRASH','UNKNOWN'),
  proof_depth           INT UNSIGNED,
  proof_length          INT UNSIGNED,
  clauses_derived       INT UNSIGNED,
  clauses_backtracked   INT UNSIGNED,
  clauses_kept          INT UNSIGNED,
  memory_allocated      INT UNSIGNED,           /* In kb */
  time                  INT UNSIGNED,           /* In seconds */
  input_time            INT UNSIGNED,           /* In seconds */
  flotter_time          INT UNSIGNED,           /* In seconds */
  inferences_time       INT UNSIGNED,           /* In seconds */
  backtracking_time     INT UNSIGNED,           /* In seconds */
  reduction_time        INT UNSIGNED,           /* In seconds */

/* Processing info */

  time_limit            INT UNSIGNED,           /* In seconds */
  memory_limit          INT UNSIGNED,           /* In kb */
  prover_parameters     BLOB,
  start_date            DATE,
```

```
  hostname              VARCHAR(255),
  machine_cpu           VARCHAR(255),
  machine_memory        INT UNSIGNED,           /* In kb */
  machine_os            VARCHAR(255),

/* Additional result info */

/* used_input_flas_nr = used_dir_refs_nr + used_bg_flas_nr */

  used_input_flas_nr    INT UNSIGNED,
  used_input_flas       BLOB,                   /* List of their names */
  used_dir_refs_nr      INT UNSIGNED,
  used_dir_refs BLOB,                   /* List of their names */
  used_bg_flas_nr       INT UNSIGNED,
  used_bg_flas          BLOB,                   /* List of their names */

/* not yet */
--  used_syms_nr                INT UNSIGNED,
--  used_syms           BLOB,                   /* Only if in the proof */

  PRIMARY KEY ( id ),
  INDEX  xproblem               (problem(20)),
  INDEX  xarticle               (article),
  INDEX  xtheorem_id            (theorem_id),
  INDEX  xprover                (prover),
  INDEX  xproof_length          (proof_length),
  INDEX  xclauses_derived       (clauses_derived),
  INDEX  xtime                  (time),
  INDEX  xstart_date            (start_date),
  INDEX  xused_input_flas_nr    (used_input_flas_nr),
  INDEX  xused_dir_refs_nr      (used_dir_refs_nr),
  INDEX  xused_bg_flas_nr       (used_bg_flas_nr)
);

/* Proof info is in separate table - it is not supposed to
   be accessed too often
*/

CREATE TABLE proof (
```

```
/* id is common with the proved table */
  id                    INT NOT NULL AUTO_INCREMENT,
  problem               VARCHAR(255),   /* Just consistency check */
  proof                 MEDIUMBLOB,     /* About 17M should be enough */
  PRIMARY KEY ( id )
);


/* Basically as proved, but some result info missing */

CREATE TABLE unproved (

/* Primary problem identification */
  id                    INT NOT NULL AUTO_INCREMENT,
  problem               VARCHAR(255),           /* Index on initial 20 chars */
  article               VARCHAR(8),             /* Article string */
  theorem_id            SMALLINT UNSIGNED,      /* Number in article */
  format                ENUM('DFG','TPTP',
                             'LOP','OTTER'),
  prover                VARCHAR(255),           /* Name and version */

/* Result info */

  result                ENUM('PROOF','COMPLETION',
                             'TIMELIMIT','MEMLIMIT',
                             'KILLED','CRASH', 'UNKNOWN'),
/*  proof_depth         INT UNSIGNED,
    proof_length                INT UNSIGNED,
*/
  clauses_derived       INT UNSIGNED,
  clauses_backtracked   INT UNSIGNED,
  clauses_kept          INT UNSIGNED,
  memory_allocated      INT UNSIGNED,           /* In kb */
  time                  INT UNSIGNED,           /* In seconds */
  input_time            INT UNSIGNED,           /* In seconds */
  flotter_time          INT UNSIGNED,           /* In seconds */
  inferences_time       INT UNSIGNED,           /* In seconds */
  backtracking_time     INT UNSIGNED,           /* In seconds */
  reduction_time        INT UNSIGNED,           /* In seconds */
```

```
/* Processing info */

  time_limit            INT UNSIGNED,          /* In seconds */
  memory_limit          INT UNSIGNED,          /* In kb */
  prover_parameters     BLOB,
  start_date            DATE,
  hostname              VARCHAR(255),
  machine_cpu           VARCHAR(255),
  machine_memory        INT UNSIGNED,          /* In kb */
  machine_os            VARCHAR(255),

/* Additional result info missing in this table */

  PRIMARY KEY ( id ),
  INDEX  xproblem               (problem(20)),
  INDEX  xarticle               (article),
  INDEX  xtheorem_id            (theorem_id),
  INDEX  xprover                (prover),
  INDEX  xclauses_derived       (clauses_derived),
  INDEX  xtime                  (time),
  INDEX  xstart_date            (start_date)
);

/* Article background info independent of prover runs */

CREATE TABLE article_bg_info (

  article               VARCHAR(8),            /* Article string */

  bg_references_nr      INT UNSIGNED,          /* complete bg theory */

  bg_dco_nr             INT UNSIGNED,          /* constructor types bg */
  bg_dco                BLOB,                  /* List of their names */
  bg_dem_nr             INT UNSIGNED,          /* mode existence bg */
  bg_dem                BLOB,                  /* List of their names */
  bg_pro_nr             INT UNSIGNED,          /* properties bg */
  bg_pro                BLOB,                  /* List of their names */
  bg_cle_nr             INT UNSIGNED,          /* cluster existence bg */
  bg_cle                BLOB,                  /* List of their names */
  bg_clf_nr             INT UNSIGNED,          /* functor cluster bg */
```

```
  bg_clf                BLOB,                   /* List of their names */
  bg_clc_nr             INT UNSIGNED,           /* condit. cluster bg */
  bg_clc                BLOB,                   /* List of their names */
  bg_dre_nr             INT UNSIGNED,           /* requirements bg */
  bg_dre                BLOB,                   /* List of their names */

  bg_syms_nr            INT UNSIGNED,
  bg_syms               BLOB,                   /* All symbols */

  PRIMARY KEY ( article ),
  INDEX xbg_references_nr       (bg_references_nr),
  INDEX xbg_dco_nr              (bg_dco_nr),
  INDEX xbg_dem_nr              (bg_dem_nr),
  INDEX xbg_pro_nr              (bg_pro_nr),
  INDEX xbg_cle_nr              (bg_cle_nr),
  INDEX xbg_clf_nr              (bg_clf_nr),
  INDEX xbg_clc_nr              (bg_clc_nr),
  INDEX xbg_dre_nr              (bg_dre_nr),
  INDEX xbg_syms_nr             (bg_syms_nr)
);
```

Appendix 3 - Manual page of the top-level problem generating script

```
MKPROBLEM(1)   User Contributed Perl Documentation   MKPROBLEM(1)

NAME
       mkproblem.pl ( Problem generating script for MPTP)

SYNOPSIS
       mkproblem.pl [options] problemnames

       mkproblem.pl -tcard_1 -trolle -ccard_2 t39_absvalue
       by_25_16_2_absvalue

        Options:
          --skipbadrefs[=<arg>],   -s[<arg>]
          --allownonex[=<arg>],    -a[<arg>]
          --basedir=<arg>,         -b<arg>
          --mml_version=<arg>,     -T<arg>
          --memlimit=<arg>,        -M<arg>
          --tharticles=<arg>,      -t<arg>
          --chkarticles=<arg>,     -c<arg>
          --filter=<arg>,          -f<arg>
          --definitions=<arg>,     -D<arg>
          --specfile=<arg>,        -F<arg>
          --allarticles,           -A
          --help,                  -h
          --man

OPTIONS
       --skipbadrefs[=<arg>], -s[<arg>]
              Skip problems with bad references.

       --allownonex[=<arg>], -a[<arg>]
              Setting to 0 causes error exit on nonexistant
              articles.
```

--basedir=<arg>, -b<arg>
        Sets the MPTPDIR to <arg>.

--mml_version=<arg>, -T<arg>
        Force mkproblem to behave as if working with MML
        version <arg>. This now influences only how the
        non-database type assertions for numerals are cre
        ated. Generally, this may influence creation of
        any formulas covering various Mizar built-in fea
        tures in the future, as they often change across
        various MML versions. The <arg> is written in the
        form 3_44_763, and for normal usage of the MPTP
        distribution, the default is OK. Use this, if you
        only want to update the MPTP scripts, without
        updating the databases to newer MML version.

--memlimit=<arg>, -M<arg>
        Sets memory limit for database caches, not imple
        mented yet.

--tharticles=<arg>, -t<arg>
        Do all theorem problems from the article <arg>,
        this option can be repeated to specify multiple
        articles.

--chkarticles=<arg>, -c<arg>
        Do all checker problems from the article <arg>,
        this option can be repeated to specify multiple
        articles.

--filter=<arg>, -f<arg>
        Specify the filtering of the background formulas.
        Default is now 1 - the checker-based signature
        filtering. Setting to 0 does no filtering at all.

--definitions=<arg>, -D<arg>
        Specify treatment of definitions. Default is now
        0 - the 'definitions' directive is neglected and

```
              definitions must be explicitely specified as ref
              erences to be included as axioms for the problems.
              1 means that all definitions from the 'defini
              tions' directive are included as direct refer
              ences.  2 is like 1, but only definitions from the
              current article are used.  3 is as 1, but the def
              initions are treated as background formulas
              instead, i.e. if filtering is activated, they are
              included only if the defined symbols appear during
              the fixpoint computation.  4 is like 3, again with
              current article definitions only.  Options 2 or 4
              are useful when we know that the definitions from
              other articles have been taken care of explicitly
              (e.g. by some previous experience like Mizar Proof
              Advisor), but this does not extend to the current
              article, which may be new in some sense.

       --specfile=<arg>, -F<arg>
              Read problem specifications from the file <arg>.

       --allarticles, -A
              Create theorem problems for all articles that are
              present in databases.

       --help, -h
              Print a brief help message and exit.

       --man  Print the manual page and exit.

DESCRIPTION
       mkproblem.pl gets a list of options and a list of theorem
       names or checker problem names (in our format, e.g.
       t39_absvalue or by_25_16_2_absvalue), and produces a com
       plete dfg file for each of them. This file contains full
       informations necessary for reproval, i.e. the references
       and all of the background info (constructor types,
       requirements, etc.).  Problem files are stored under their
       articles directories, in the PROBLEMS directory.
```

CONTACT

Josef Urban urban@kti.ms.mff.cuni.cz