Contents

Editorial

This report contains the proceedings of the IJCAR 2004 Workshop 7 on *Computer-Supported Mathematical Theory Development*, held July 5, 2004 in Cork, Ireland.

Mathematical reasoning tools, such as computer algebra systems, theorem provers, decision procedures, etc., are increasingly employed in mathematics and engineering. Also large repositories of formalized mathematics are currently emerging. It is nevertheless the case that the actual pragmatics of mathematics is still to be characterized as mainly pen and paper based. One reason is that still no convincing systems exist that provide a sufficiently integrated support for the usual work phases of a mathematician, e.g. from initial conception and organization of ideas up to the final publication in a journal article.

A special focus of the workshop is on computer-support for the development of mathematical theories. Mathematical theory development describes the formulation, organization, manipulation, and maintenance of mathematical content. Support for adequate interaction with the (human) mathematician is mandatory in this context.

Thus, computer-supported mathematical theory development comprises:

- the formulation of mathematical statements in a computer-processable form,
- computer-support in processing mathematical content; depending on the content, this can mean "proving", "computing", "solving", "visualizing", "checking", "simulating", "conjecturing", etc.
- the systematic organization and maintenance in and the powerful retrieval of mathematical knowledge from computer-accessible media,
- the management of change in the development of mathematical knowledge,
- the publication and presentation of mathematical material using new and/or well-established computer-based publication or presentation formats, and
- the interaction between the human mathematician and the supporting software.

The workshop addresses the design and implementation of frameworks aiming at integrated support for the entire process of theory development. Clearly, there is still a big gap between the systems envisioned and the systems already available and this gap has to be overcome in the future. Therefore also partial solutions are welcome if their relevance for the bigger vision can be illustrated.

The workshop's program committee:

Michael Beeson, San Jose, USA	Dieter Hutter, Saarbrücken, Germany
Bruno Buchberger, Linz, Austria	Christoph Kreitz, Potsdam, Germany
Alan Bundy, Edinburgh, Scotland	Volker Sorge, Birmingham, England
Simon Colton, London, England	Andrzej Trybulec, Bialystok, Poland
William Farmer, Hamilton, Canada	Freek Wiedijk, Nijmegen, The Netherlands

The organizers want to thank all members of the program committee for their valuable help during the refereeing process so that we managed to get along with the tight schedule.

We also want to thank the IJCAR'04 Workshop Chair, Peter Baumgartner, for his help in all organisational matters.

The organizers are grateful to Larry Paulson for agreeing to give an invited talk on *Formalizing Abstract Mathematics: Issues and Progress.* Finally, we thank CoLogNET and the European Union CALCULEMUS RTN for supporting our workshop.

June 2004 **Christoph Benzmüller and Wolfgang Windsteiger**

Formalizing Abstract Mathematics: Issues and Progress

Larry C. Paulson
University of Cambridge, UK

Invited Talk

Abstract

Many well-known theorems in mathematics have been formalized using a variety of tools and formal systems. Despite this success, some kinds of material remain difficult to formalize, especially if our aim is to develop an ever-growing book of mathematics rather than stopping once a certain theorem has been proved. Algebra, with its numerous abstract concepts that are refined and combined in countless ways, is particularly difficult to formalize, but other branches of mathematics, such as topology, present similar difficulties. The speaker will describe recent progress in this area using Isabelle's locale construct.

Towards a Generic Management of Change

DIETER HUTTER
GERMAN RESEARCH CENTER FOR ARTIFICIAL INTELLIGENCE
(DFKI GMBH)
STUHLSATZENHAUSWEG 3, 66123 SAARBRÜCKEN, GERMANY
E-MAIL: HUTTER@DFKI.DE

Abstract

In this paper we sketch the outline and the underlying theoretical framework for a general repository to maintain mathematical or logic-based documents while keeping track of the various semantical dependencies between different parts of various types of documents (documentations, specifications, proofs, etc). The sketched approach is a generalization of the notion of development graphs (as implemented in the MAYA-system) used to maintain formal software developments. We isolate maintenance mechanisms that solely depend on the structuring of objects and their relations. These mechanisms define the core of the general repository while mechanisms that are specific to individual semantics are sourced out to individual plug-ins attached to the general system.

1 Introduction

It is well-known that the logical formalization of software systems is error-prone. Since even the verification of small-sized industrial developments requires several person months, specification errors revealed in late verification phases pose an incalculable risk for the overall project costs. An *evolutionary formal development* approach is absolutely indispensable. In all applications so far, development steps turned out to be flawed and errors had to be corrected. The search for formally correct software and the corresponding proofs is more like a *formal reflection* on partial developments rather than just a way to assure and prove more or less evident facts. In the same way mathematical knowledge has to be constantly revised, modified and enlarged. Discovering mathematical knowledge is a continuous interplay between (re-)specifying mathematical structures and proving proposed properties about these created structures.

In this paper we aim at a repository to maintain all sorts of dependencies between various parts of a formal development. The main goal of such a logic-based repository is to ease the development of mathematical or logic based knowledge consisting of entities such as axioms, definitions, theorems, proofs or even programs or informal documentations. As the development of a software project or of mathematical knowledge is distributed, also the repository has to support distributed developments. A CVS-like infrastructure [2] is necessary to determine the differences between two versions, to calculate the necessary changes to update a local repository to the current state, and to integrate two rival developments into a merged variant. It maintains the various dependencies between individual mathematical entities and keeps the database always in a consistent state, calculating the effects of changes in the database and using logic based tools to adjust existing proofs once the underlying theory has changed. We aim at a repository that encompasses a management of change maintaining these dependencies and suggesting necessary adjustments after changing mathematical objects.

Mathematical proofs are the central entities in the logic based repository but they are also the most fragile ones. Changes in the repository will always endanger the stored proofs as assumptions used in these proofs may have changed. While the repository has to recognize those situations in which changes did not affect the assumptions of a proof, there is a need for techniques to support the reuse of proofs if parts of the assumptions have been changed, because, for instance, some theorems and proofs have been transferred from one theory to another or theories have been merged or split. We aim at a machine assistance to recreate a proof corpus following a change in the axioms/definitions. The re-creation would be by a combination of: certifying which proofs are unaffected by the change, adapting other proofs by analogy into new proofs, simplifying proofs when possible, and user interaction to complete any residual partial proofs.

2 MAYA

Within the last years we developed a stand-alone repository, MAYA, supporting the management of change in the domain of algebraic specifications. The MAYA-system [5, 1] allows users to specify and verify developments in a structured manner, incorporates a uniform mechanism for verification *in-the-large* to exploit the structure of the specification, and maintains the verification work already done when changing the specification. MAYA relies on development graphs as a uniform representation of structured specifications, which enables the use of various (structured) specification languages like CASL [3, 9] to formalize the software development. While unstructured specifications are solely represented as a signature together with a set of logical formulas, the structuring operations of the specification languages (such as `then`, `and`, or `with` in CASL) are translated into the structure of a development graph. Each node of this graph corresponds to a theory. The axiomatization of this theory is split into a local part which is attached to the node as a set of higher-order formulas and into global parts, denoted by ingoing definition links, which import the axiomatization of other nodes via some consequence morphisms. While a so-called *local* link imports only the local part of the axiomatization of the source node of a link, *global* links are used to import the entire axiomatization of a source node (including all the imported axiomatizations of other nodes). In the same way local and global *theorem links* are used to postulate relations between nodes (see [5] for details). As theories correspond to subgraphs within the development graph, a relation between different theories, represented by a global theorem link, corresponds to a relation between two subgraphs. Each change in these subgraphs can affect this relation and would invalidate previous proofs of this relation. Therefore, MAYA decomposes relations between different theories into individual relations between the local axiomatization of a node and a theory (denoted by a local theorem link). Each relation decomposes again into a set of proof obligations postulating that each local axiom of the source node is a theorem in the target theory with respect to the morphism attached to the link.

To this end MAYA provides a generic interface to plug in additional parsers for the support of other specification languages. Moreover, MAYA allows the integration of different theorem provers to deal with the actual proof obligations arising from the specification, i.e. to perform verification *in-the-small*.

3 A General Approach — The System Design

When developing repositories for various purposes, like e.g. formal developments (MAYA [1]) or course material for Universities (MMISS [7]) it turns out that maintaining semantic dependencies between all the various documents is one of the main challenges. To support a distributed development, the repository should compute differences between different branches of a development and should be able to merge different branches into a common one. However, in contrast to CVS, the repository should be aware of semantic dependencies and notify conflicts if such semantic dependencies are violated when manipulating the development. However, the vast number of such dependencies occurring in practical examples prohibits the brute-force control of each individual dependency each time some document has been edited. Tracking dependencies in developments requires a structuring of dependencies. Typically, there are different sources for such dependencies:

On the one hand the semantics of (structured) objects depends on the semantics of objects used for their definition (axiomatic dependencies). While there is no way to scrutinize changes of axiomatic dependencies in case of intentional (or purposeful) changes of the development, there is a need to inspect axiomatic dependencies in case of mechanical changes as they occur during the (automatic) merge of two branches. Section 4 will discuss this issue in more detail.

On the other hand properties between (structured) objects can be postulated and proven within a development. Similar to the objects under consideration, the proofs of properties about such objects are also structured. Hence, such a proof depends on (or decomposes into) properties of sub-objects which gives rise to *deduced dependencies* between different properties. Changing the development may render proofs invalid since either some basic property does no longer hold or the way the problem was decomposed is no longer appropriate. The decomposition of proof obligations according to the structure of the objects allows us to reuse already established properties of those sub-objects that have not been changed. Section 5 illustrates a generic mechanism that makes use of this observation to minimize the effect of changes to proofs of properties.

4 Distributed Developments

The development of mathematical knowledge bases or formal (software) developments is usually distributed over various developers working on different parts of the actual development. As a consequence there are various versions of the development at different places differing in the fields of activity of the individual mathematicians or engineers. At some points of the development process these concurrent progresses have to be merged into a common document.

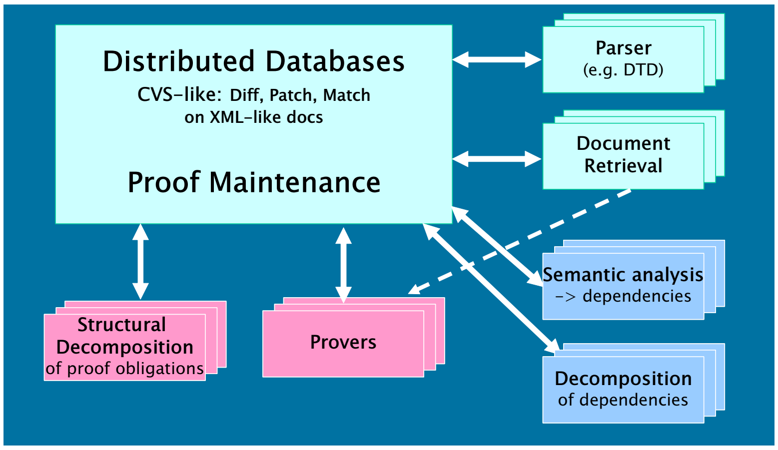


Figure 3.1: Distributed Development according CVS

CVS (Concurrent Versions Systems) [2] (`http://www.cvshome.org`) has been well-known for supporting a concurrent development of text documents. The idea of CVS is to keep all documents in a common repository. In order to work on documents, a user has to create a local copy of the repository. Independently of other developers she can change the documents in the local copy of the repository in an arbitrary way. Eventually she decides to make her changes available to all other users, which means that she wants to integrate her changes into the common repository. If the actual version of the common repository is still the version she used to start her modifications then CVS can simply replay all the changes the user made locally also in the common repository. However, suppose there is another user who changed parts of the document and committed her changes to the common repository first. In this case the common repository is not identical to the starting point of the first user and CVS has to *merge* the changes of both users into a common development. CVS considers documents as sequences of text lines. The change of a document is *decomposed* to changes of text lines. A so-called conflict occurs if the same text line has been independently altered (changed or removed) by both users in different ways. In this case the user has to decide which version she will prefer. In all other cases the newest version of the text line will be used or removed respectively, if one user removed the line in her version.

While text-lines might be appropriate to structure pure text documents, this approach fails completely in logic-based documents for the following reasons: First, line-feeds do not correspond to the intended structure of logic-based document which are usually described by a term-based language. A single text line may contain two independent terms which could be changed independently. Moreover, a single term could be spread over many text lines and undiscovered "semantic" conflicts may occur if two users change different text lines that are both part of the description of a single term. Second, there are usually many "semantic" dependencies between different parts of the document. Changing the arity of a signature symbol in a document typically requires to change its arity in all the occurrences of the symbol.

In the following, we reuse the general paradigm of CVS consisting of computing

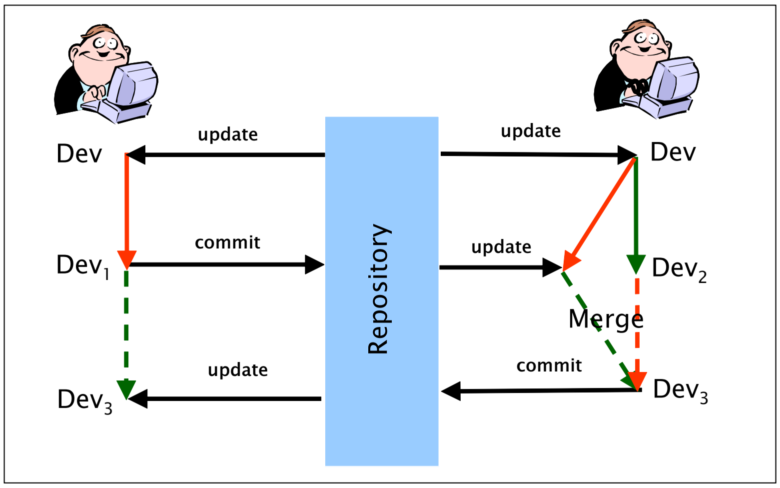


Figure 4.2: Distributed Development According CVS

differences or patches and merging different threads of developments by automatically joining independent changes while asking the user to resolve competing changes. However, instead of using the physical structure of documents as a sequence of text lines we will use more semantically oriented structures in a first phase. In the second phase we add semantical dependencies between different parts of a document to detect conflicts between different versions of a document that result from changing different but semantically still dependent parts of a document by different users.

Languages like XML have been advertised as a common standard to formulate structured knowledge. Within this approach we follow this idea and assume documents to be described in an XML-like syntax. Therefore documents are usually trees, the nodes of which are attached with information. For instance, terms may be represented in XML-like languages (e.g. MathML `http://www.w3.org/Math/`) basically by their corresponding term tree. Nodes are annotated with signature symbols and the subtrees of the node refer to the arguments of the symbol. We generalize this concept and allow also structure sharing which results in the notion of acyclic directed graphs as the general structure underlying the documents under consideration[1]:

Definition 1 *A* **structured document** $O = \langle \mathcal{N}, \mathcal{L} \rangle$ *is an acyclic directed graph consisting of* **nodes** $\mathcal{N}$ *and* **containment links** $\mathcal{L}$. $\mathcal{O}$ *is the set of all structured documents. A* **syntax parser** ϕ *is a predicate on* $\mathcal{O}$ *and* $\{O \in \mathcal{O} \mid \phi(O) \text{ holds}\}$ *is the set of structured documents admissible wrt.* ϕ.

The notion of admissibility in Definition 1 refers to syntactical restrictions of the documents which are typically covered, for instance in XML, by using DTDs or XML-Schemas. In the following, we interprete this notion of admissibility as additional restrictions to acyclic directed graphs in order to denote well-formed documents. However, our proposed approach is orthogonal to the definition of admissibility and therefore we will not explore its internal representations.

[1] XML already provides some notion of structure sharing by the notions of ID and IDREF.

Let $N, M \in \mathcal{N}$ and $\mathcal{L}$ be a set of links. Then we write $N \rightarrow^*_{\mathcal{L}} M$ iff $N = M$, or there is a $N' \in \mathcal{N}$ such that $N \rightarrow N' \in \mathcal{L}$ and $N' \rightarrow^*_{\mathcal{L}} M$. Let $O = \langle \mathcal{N}, \mathcal{L} \rangle$ and $N \in \mathcal{N}$. Then $O_N = \langle \mathcal{N}', \mathcal{L}' \rangle$ is the smallest graph with $\mathcal{N}'$ being those nodes of $\mathcal{N}$ from which N is reachable and $\mathcal{L}'$ is the subset of links in $\mathcal{L}$ which are on a path between a node of $\mathcal{N}'$ and N.

We assume that individual nodes or links may contain individual content. Nodes and links with different content are different. This means later-on that changing the content of a node is formally an insertion of a new node to the graph inheriting all links from the old node. Similar remarks hold for links.

In order to implement a CVS-like repository we have to provide appropriate definitions

- to compute the *differences* of two structured documents,
- to operationalize this description as a *patch* for transforming one structured document into another, and
- to merge two different patches of a common structured document to a common patch including as much as possible of both patches.

Differences between two structured documents can be easily described by the differences of their corresponding parts, i.e. set of nodes and sets of links. Then patches are sequences of insertions or deletions of nodes or links. Notice, that this abstract mathematical view of documents as structured documents hides some practical issues of comparing graphs which are related to the question on when two nodes of different documents or patches are considered to be equal (or identical). We argue that introducing unique identifiers for nodes will allow us to identify corresponding nodes in different versions of a document (see [4]).

Defining the difference between two structured documents by $O_1 \setminus O_2 := \langle \mathcal{N}_1 \setminus \mathcal{N}_2, \mathcal{L}_1 \setminus \mathcal{L}_2 \rangle$[2], etc, we can define the merge of two structured documents wrt. to a common predecessor document formally:

Definition 2 *Let O, O_1, and O_2 be three structured documents. Then, the merge of O_1 and O_2 wrt. a common predecessor O is defined by*

$$merge(O_1, O_2, O) := (O \cap O_1 \cap O_2) \cup (O_1 \setminus O) \cup (O_2 \setminus O)$$

It is easy to see that $merge(O_1, O_2, O) = (O_1 \setminus (O \setminus O_2)) \cup (O_2 \setminus (O \setminus O_1))$.

Definition 3 *A merge $merge(O_1, O_2, O)$ is* **admissible** *iff $merge(O_1, O_2, O)$ is an admissible object.*

This admissibility check refers again to the syntactical restrictions of structured documents given, for instance, in a DTD or an XML-Schema. In case we use unique identifiers to detect corresponding nodes in different versions we also have to check the uniqueness of these identifiers in order to guarantee that the merge does not contain two nodes

[2] Notice that the difference of two structured documents is in general not a structured document since there might be links in $\mathcal{L}_1 \setminus \mathcal{L}_2$ which refer to nodes that are not member of $\mathcal{N}_1 \setminus \mathcal{N}_2$. Here, we use this notion simply as a pair of sets.

containing competing information. In this case we have a conflict arising by the change of the same node by two developers. Thus, the user has to decide which of these nodes has to be selected for the merged version.

4.1 Semantic Dependencies

Given a structured document, which is usually a syntactical object, its parsing will reveal an internal semantic structure. Partly this structure can be explicitly given by syntactical means. Consider for instance the example given in Figure 4.3.[3] We specify lists of natural numbers `List of Nats` with the help of the specification of `Nats` which is syntactically indicated by the reference `Imports: Nats`. Looking at the use of `Nats` in `List of Nats` more closely, we recognize that symbols like $\leq$, 0 or *succ* are used which are defined in `Nats` which gives rise for various dependencies between the definitions and the uses of these symbols.

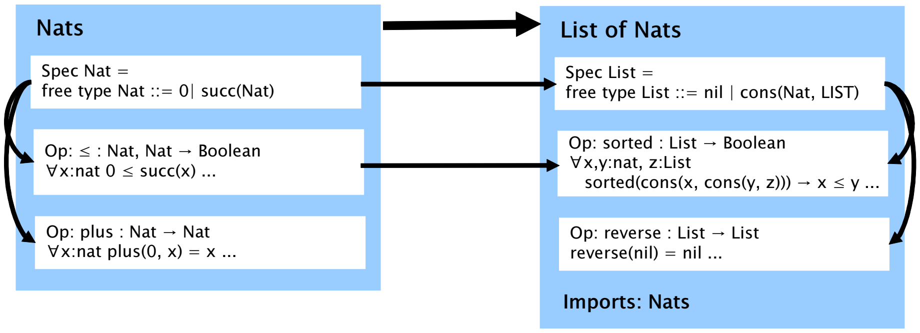


Figure 4.3: Semantic Dependencies in Specifications

Thus, we extend structured documents by a new kind of link which represents the semantic dependencies between (different) objects. The implicit understanding of these links is that the user is not free to define them arbitrarily but they are computable in a uniform way from a corresponding structured document. Furthermore we are interested to localize the necessary part of the object that have to be considered when computing the semantic dependencies between two individual subgraphs.

As mentioned before, semantic dependencies occur on various levels of the structured document (considered as a syntactical representation of a structured semantical object). For instance, there are dependencies between theories which are reflected by the dependencies of signature symbols defined in one theory but used in another. Hence, we will introduce a refinement mapping that allows us to decompose a dependency between structured documents to individual dependencies of their sub-objects.

Definition 4 *An* **extended** *structured document O is a tuple $\langle \mathcal{N}, \mathcal{L}, \mathcal{U}, \mathcal{R} \rangle$ such that*

[3]In this simple example, the boxes denote individual nodes, the nesting of the boxes represents the containment links $\mathcal{L}$ while semantic dependencies are given by links between different boxes.

$\langle \mathcal{N}, \mathcal{L} \rangle$ is a structured document, $\mathcal{U}$ is a set of **dependency** *links between nodes in $\mathcal{N}$ and $\mathcal{R}$ is a mapping from elements of $\mathcal{U}$ to subsets of $\mathcal{U}$ such that*

- *$\langle \mathcal{N}, \mathcal{L} \cup \mathcal{U} \rangle$ is an acyclic graph.*
- *for all $N \to M \in \mathcal{U}$ with $\mathcal{R}(N \to M) = \{N_1 \to M_1, \ldots N_n \to M_n\} \subseteq \mathcal{U}$ $N_i \to^*_{\mathcal{L}} N$, and $M_i \to^*_{\mathcal{L}} M$ holds.*

The first condition of the definition guarantees that there are no cyclic dependencies considering the existing containment relation. The second condition restricts the way a dependency can be decomposed. In principle the decomposition of a dependency of two structured documents should result in dependencies of individual sub-objects of the considered structured documents.

Definition 5 *A* **semantic analyzer** *is a partial mapping φ of structured documents to extended structured documents such that for all $\mathcal{N}$, $\mathcal{L}$ for which φ is defined there are $\mathcal{U}$ and $\mathcal{R}$ with $\varphi(\langle \mathcal{N}, \mathcal{L} \rangle) = \langle \mathcal{N}, \mathcal{L}, \mathcal{U}, \mathcal{R} \rangle$.*

A set of dependency links represents the semantic dependencies between different parts of a structured document. Intuitively, a merge of two structured documents causes a conflict if there would be a dependency between two objects which stem from different branches. Since in this case both objects are manipulated separately by different developers, none of them has anticipated the resulting interplay between both structured documents.

Hence, analyzing the merged structured document and inspecting the set of arising dependency links $\mathcal{U}'$ we demand that for each dependency link either the *same* link connecting the *same* structured documents existed in one of the original structured documents or that the link is decomposable, i.e. there is a refinement that decomposes the link l to a set of more fine-grained dependency links $\mathcal{R}(l)$ in $\mathcal{U}'$ which all satisfy this condition.

As an example consider two developers D_1 and D_2 working simultaneously on the specification given in Figure 4.3. While developer D_1 detects a specification error inside the definition of `reverse` and corrects its definition in `List of Nats`, developer D_2 exchanges the definition of the predicate $<$ by some definition for $>$ and adjusts the definition of the predicate `sorted` accordingly. When we merge the branches of both developers we obtain in particular an emerging version of `List of Nats` which did not exist in any of the two branches. This potential conflict can be resolved if we refine the dependency between `Nats` and `List of Nats` to dependencies between their sub-objects. Since the definition of reverse is independent of the specification of `Nat` and `sorted`, the changes made by the two developers operate on independent subgraphs.

Using the following abbreviations $\varphi(O) = \langle \mathcal{N}, \mathcal{L}, \mathcal{U}, \mathcal{R} \rangle$, $\varphi(O') = \langle \mathcal{N}', \mathcal{L}', \mathcal{U}', \mathcal{R}' \rangle$, etc, we define:

Definition 6 *Let O_1, O_2, O be structured documents, φ be a semantic analyzer, and $O' = merge(O_1, O_2, O)$ with $\varphi(O') = \langle \mathcal{N}', \mathcal{L}', \mathcal{U}', \mathcal{R}' \rangle$. Then a link $l : N \to M \in \mathcal{U}'$ is* **valid** *wrt O_1, O_2, O and φ iff*

- *O'_N, O'_M are subgraphs of O_1 and $l \in \mathcal{U}_1$,*

- O'_N, O'_M *are subgraphs of* O_2 *and* $l \in \mathcal{U}_2$*, or*
- *The set* $\mathcal{R}'(l)$ *is not empty and all dependency links* $l' \in \mathcal{R}'(l)$ *are valid wrt.* O_1, O_2, O *and* φ*.*

Definition 7 *Let* O_1, O_2, O *be structured documents,* φ *be a semantic analyzer, and* $O' = merge(O_1, O_2, O)$ *with* $\varphi(O') = \langle \mathcal{N}', \mathcal{L}', \mathcal{U}', \mathcal{R}' \rangle$*. Then,* O' *is* **valid** *wrt. wrt* O_1, O_2, O *and* φ *iff all* $l \in \mathcal{U}'$ *are valid wrt.* O_1*,* O_2*,* O *and* φ*.*

Structuring dependency links in a hierarchical way, makes it feasible to track dependencies in an efficient way. The idea is to inspect the validity of dependency links along the containment-relations of the structured documents. First, we analyze all dependency links l which are not part of the refinement of any other dependency link. Links in $\mathcal{R}(l)$ are only inspected if the first two cases in Definition 6 do not hold. To implement such an approach we have to demand additional requirements to the semantical analyzer. The essential feature is the property that the analyzer always creates the same refinement for a link l if the connected structured documents have not changed. However, there are cases in which specification languages violate this property. The reason is that a so-called global environment is used to assign the occurrence of a symbol to its definition. If there are multiple definitions of the same symbol then a change of the environment might cause different assignments of unchanged objects. For example, suppose that `List of Nats` would import another theory and we would add a definition of $\leq$ to this theory. This new definition may hide the definition of $\leq$ in `Nats` and thus eliminate the dependency link between that definition and the definition of `sorted`. It is ongoing work to find appropriate properties of the semantic analyzer also for these situations which will allow us to avoid unnecessary checks of dependency links.

5 Maintaining Structured Properties

Inspecting the ideas of MAYA we discovered that most of the work related to the management of change does not require a deep knowledge of the semantics of the underlying specification languages. Instead the management of change solely operates on the structure of the objects under consideration and on how proposed properties can be decomposed to properties of their sub-objects. To cope with postulated or verified properties we introduce another type of links $\mathcal{P}$ denoting (derived) properties between structured documents. Similar to the refinement (or decomposition) of dependency links in $\mathcal{U}$ we like to decompose properties (i.e. links in $\mathcal{P}$) to corresponding properties of involved sub-objects. Therefore we introduce a mapping $\mathcal{D}$ which is a partial function decomposing links in $\mathcal{P}$ to sets of links in $\mathcal{P}$.

Consider the development in Figure 5.4. We have specified independently `List of Nats` and `Generic List` and would now like to prove that `List of Nats` includes[4] `Generic List` (denoted by link No. 0) wrt. some signature morphism (omitted here for sake of simplicity). The semantics of development graphs allow us to decompose this proof obligation into various other obligations (denoted by all the links annotated with 1): we have to prove that all axioms of `Generic List` are theorems in `List of`

[4]in terms of theory inclusion.

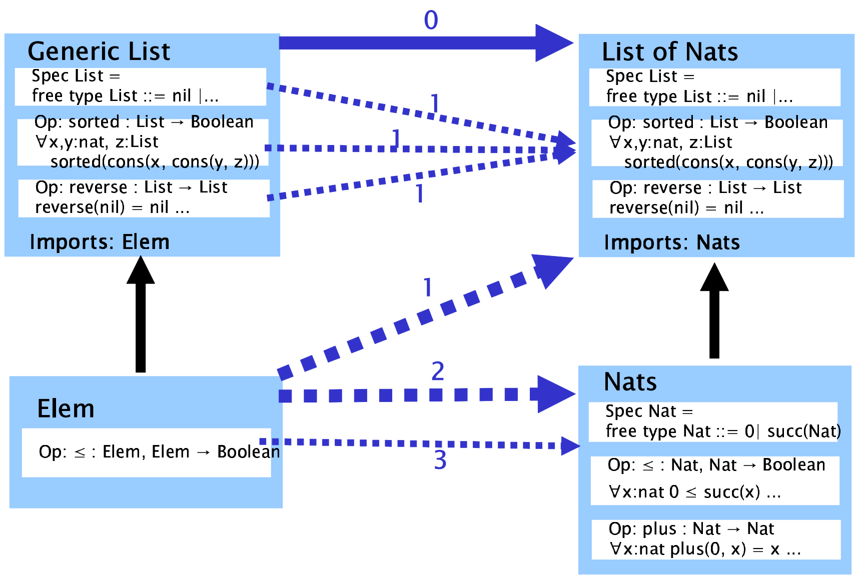


Figure 5.4: Decomposition of Proofs

`Nats` and that `Elem` is included in `List of Nats`. The latter proof obligation can be decomposed again into the theory inclusion property between `Elem` and `Nats` (indicated by link No. 2) and which itself is decomposed into the property of the axiom of `Elem` being a theorem in `Nats` (link No. 3).

Decomposition of proof obligations minimizes the effects of changing parts of the development locally. If we only change the axiom of `Elem` then we only have to reestablish the proof that this axiom (mapped via some signature morphism) is a theorem in `Nats`. Without decomposition we would have to prove the property that `Generic List` is included in `List of Nats` from scratch. If we change `Generic List` by adding another theory then we have to redo the decomposition of link 0 resulting in an additional proof obligation that the new theory is also included in `List of Nats`. All other proofs are not affected.

The ability to decompose properties along the structure of the concerned objects allows us to localize the effects of changes. A property between structured objects (theories) is decomposed (according to some decomposition rules) to properties between their sub-objects (local axioms). Typically these properties between structured objects are independent of the environment in which these objects might occur. As long as the concerned structured objects are unchanged any change of the overall development will not inflict the already proven or postulated properties between these objects. Hence, the result of decomposing a link in $\mathcal{P}$ only depends on that link and the subgraphs connected by the link.

The crucial observation is that in order to compute these decompositions we only need shallow information about involved structured documents. To decompose link 0 in our example we do not have to inspect the internal structure of the axioms of `List of`

`Nats`, nor do we have to know anything about the internal structure of `List of Nats`. This means that the computed decomposition is still valid when we change an individual axiom of `Elem` or import an additional theory to `List of Nats`. Hence, in the proposed system each decomposition is associated with a subset of nodes and links which are used to justify the decomposition. Only if this particular part of graph is changed then we have to recompute the decomposition and adjust the graph accordingly.

The system itself does not require any knowledge about how to compute the decomposition. This can be done by an external procedure which – given a specific proof obligation – will return the new set of proof obligations (encoded again as links in the extended structured document) and the set of nodes used to compute the decomposition. The system itself will store this information and will retain the computed decomposition as long as the used set of nodes indicated by the procedure does not change.

6 Conclusion

We sketched the outline and some basic notions of a generic tool to support the distributed development of structured documents including postulated or verified relations among them. The key observation is that the mechanisms used to realize a distributed management of change mainly depend on the structuring mechanisms (rather than on the individual semantics of the basic objects) used within the applications. In order to cope with the growing complexity when dealing with realistic examples we introduced various notions of decompositions which allow for a hierarchical management of change.

The ultimate goal is to support generic structuring mechanisms as they occur in various domains by developing a system supporting these mechanisms while outsourcing application specific parts into modules attachable to the system. This would allow us to instantiate such a system for various purposes, like for instance in formal methods (cf. MAYA [1]), program development, or even maintaining course materials (cf. MMISS [7]).

Bibliography

[1] S. Autexier, D. Hutter, T. Mossakowski and A. Schairer. The Development Graph Manager MAYA. In Proceedings 9th International Conference on Algebraic Methodology And Software Technology, AMAST2002. Springer, LNCS 2422, 2002

[2] B. Berliner. CVS II: Parallelizing Software Development. In Proceedings of the USENIX Winter 1990 Technical Conference, USENIX Association, 1990

[3] M. Bidoit and P.D¿ Mosses. CASL User Manual: Introduction to Using the Common Algebraic Specification Language. Springer, LNCS 2900, 2004

[4] P. Buneman, S. Khanna, K. Tajima, W.C. Tan. Archiving Scientific Data. In Proceedings of ACM SIGMOD International Conference on Management of Data, 2002

[5] D. Hutter. Management of Change in Verification Systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, IEEE Computer Society, 2000.

[6] M. Kohlhase and R. Anghelache. Towards Collaborative Content Management and Version Control. Proceedings of the 2. International Conference on Mathematical Knowledge Management, MKM 2003, Springer, LNCS 2594, 2003

[7] B. Krieg-Brückner, D. Hutter, C. Lüth, E. Melis, A. Pötsch-Heffter,

M. Roggenbach, J. Smaus and M. Wirsing. Towards MultiMedia Instruction in Safe and Secure Systems. In: Recent Trends in Algebraic Development Techniques, (WADT-02). Springer, LNCS 2755, 2003

[8] B. Krieg-Brückner and J. Peleska and E.-R. Olderog and A. Baer. The UniForM Workbench, a Universal Development Environment for Formal Methods. In J. M. Wing and J. Woodcock and J. Davies (eds): Proceeding of FM'99 - Formal Methods. Springer, LNCS 1708, 1999

[9] T. Mossakowski and P. Hoffman and S. Autexier and D. Hutter. Part IV: CASL Logic. In B. Krieg-Brückner and P. Mosses (eds): The CASL Reference Manual, Springer, LNCS 2960, 2004

An Environment for Building Mathematical Knowledge Libraries

Florina Piroi[1] Bruno Buchberger

Research Institute For Symbolic Computation,
4232 Hagenberg, Austria

{florina.piroi, bruno.buchberger}@risc.uni-linz.ac.at

Abstract

In this paper we identify the organizational problems of Mathematical Knowledge Management and describe tools that address one of these problems, namely, the additional annotation of formalized knowledge. We describe, then, how the tools are realized in the frame of the Theorema system.

1 Introduction

The aim of the new research area "Mathematical Knowledge Management" (MKM) is the computer-support (partial or full automation) of all phases of the exploration of mathematical theories:

- invention of mathematical concepts,
- invention and verification (proof) of mathematical propositions,
- invention of problems,
- invention and verification (correctness proofs) of algorithms that solve problems,

and the structured storage of concepts, propositions, problems, and algorithms in such a way that, later, they can be easily accessed, used and applied.

MKM in this broad sense is an essentially logical activity: All formulae (axioms and definitions for concepts, propositions, problems, and algorithms) must be available in the coherent frame of a logical system, e.g. some version of predicate logic and the main operation of MKM on these formulae is essentially formal reasoning (in particular formal proving).

The *Theorema* system is one of the systems whose emphasis is on this logic aspect of MKM, which we think is the fundamental aspect of future MKM. Some papers on the logical aspects of MKM within *Theorema* are [12, 5]. The question of computer-supported invention of mathematical knowledge within *Theorema* is treated in [9], the question of computer-supported algorithm synthesis within *Theorema* is treated in [6, 8] and [10].

On the surface of MKM, however, we are faced also with many additional organizational problems, which are important for the practical success of MKM:

[1]partially supported by the RISC PhD scholarship program of the government of Upper Austria and by the FWF (Austrian Science Foundation) SFB project P1302.

a. The translation of the vast amount of mathematical knowledge which is available only in printed form (in textbooks, journals etc.) and which has to be brought into a form (e.g. LaTeX), in which it can be processed by computer algorithms. This is the problem of "digitization" of mathematical knowledge, see e.g. [25] for a survey on the existing projects in this area. The *Theorema* project is not engaged in this area of MKM.

b. The translation of digitized mathematical knowledge, for example in the form of LaTeX files, into the form of formulae within some logical system, e.g. predicate logic so that, afterwards, they can be processed by reasoning algorithms (in particular automated theorem provers). Many current projects are addressing this question, see e.g. MathML [26], OpenMath [13]. The *Theorema* project is not engaged in this area of MKM.

c. The organization of big collections of formulae, which are already completely formalized within a logic system (e.g. predicate logic) in "hierarchies of theories". At the moment, the largest such collection is Mizar [21]. Among other existing ones we mention MBase [19], the Formal Digital Library project [1], the NIST Digital Library of Mathematical Functions [20], the libraries of the theorem provers Isabelle [17], PVS [23], IMPS [16], Coq [14].

   The subproblem c., again, has two sub-aspects:

   c1. The organization of formalized mathematical knowledge by means of mathematical / logical structuring mechanisms like domains, functors, and categories. (Within *Theorema*, these questions are treated, for example, in [7].)

   c2. The additional annotation of formalized mathematical knowledge by "labels", so that blocks of mathematical knowledge can be identified and combined in various ways without actually going into the "semantics" of the formulae.

For the above and other overall views of MKM see [11, 2] and [4].

This paper exclusively deals with the subproblem c2. Traditionally, mathematical texts (collections of formulae) are organized in chapters, sections, subsections, etc. and individual formulae may have additional descriptive key words like "Definition", "Theorem", "Lemma" etc. and subformulae may also have individual labels like "(1)", "(2)", "(a)", "(b)", or "(associativity)" etc. All these external descriptors of formulae are used as (hierarchical) labels, which have no actual logical meaning or functionality, but they are only used for quick (and hopefully unique) referencing of formulae in big mathematical texts. Also, parts of large mathematical texts may be available in various files and often we will like to include text from various files as parts of another.

In traditional mathematical texts, these various descriptors of blocks of formulae and individual formulae are usually assigned in an ad hoc way. However, for the future computer-based management of mathematical knowledge, tools for generating and using these descriptors for accessing pieces of mathematical text and individual formulae are of vital practical importance.

In this paper, we report on tools which we developed recently for supporting the automated generation of unique labels (descriptors) for formulae and collections of formulae within the *Theorema* system and for using these labels in a systematic way for the build-up of coherent formal mathematical texts, i.e. collections of formulae within the *Theorema* version of predicate logic. Although these tools have been developed for *Theorema*, the design principles of the tools are independent of *Theorema* and may be useful also for other systems of formal mathematics. The design of the tools is based on ideas of the second author, the concretization for *Theorema* and actual implementation is part of the first author's forthcoming PhD thesis [24].

The plan of the paper is as follows: In section 2 we review the work that is going on in the area of Mathematical Knowledge Management and we give the main design idea of our tools, as a mathematical document editing environment. In section 3 we will describe how they are integrated in *Theorema*. We will end with conclusions and remarks on future work in section 4.

2 Towards Mathematical Document Parsing

When thinking of a mathematical knowledge base, most of us will, more or less, have in mind a big collection of formulae (definitions, theorems, etc.) organized in some hierarchical structure. Usually, this knowledge is to be found in specialized books, which have the big disadvantage of presenting the information in a static way. Searching in them can only be done syntactically and is time consuming. An important step forward was done by using computers to electronically store and search within mathematical documents (organizational problem a. in the previous section).

As the Internet became one of the most handy and used tools for finding information, it was a natural step to employ it for making mathematical knowledge widely available. Still, for some time, mathematical formulae were displayed only as graphics.

Using the MathML recommendation of W3C [26], it is now possible to display and communicate formulae. Being an application of XML, MathML benefits from the existing tools that manipulate XML files. Though it does offer some semantics of the symbols in the mathematical formulae, the set of these symbols is too restricted when compared to those used by working mathematicians. To ameliorate this situation projects like OpenMath [13] and OMDoc [18] emerged. The OpenMath standard concentrates on representing mathematical objects together with their semantic meaning, allowing them to be exchanged between computer programs, stored in databases, or published on the worldwide web. Though it is not exactly so, one can view OpenMath as extending the MathML capabilities by using "content dictionaries" where mathematical symbols are defined syntactically and semantically. OMDoc is an extension of OpenMath and MathML, adding capabilities of describing the mathematical context of the used OpenMath objects.

An important drawback of the standards mentioned above is that the coherence of the different documents (e.g. content dictionaries) is not automatically checked. This has to be done by a human, the task being rather difficult because the representation formats are not human oriented. This representation confronts us with another issue, which we intend to address in this paper: publishing mathematics using these representations is not attractive for the everyday mathematician. There is ongoing work to improve this state of facts, the latest the authors are aware of being presented in [15].

The mathematical documents that a user types into the computer are one main ingredient in building a mathematical knowledge base. In the ideal case, the human user does nothing else than typing his or her ideas and formulae into the computer, using a user-friendly environment that allows easy formula editing, like Maple or Mathematica. A program will take, then, this document and process it, extracting all the information of interest, organizing it, correlating it with (eventual) existing documents, making it available for the theorem provers, maybe even correcting eventual typos. As this is at the moment not yet possible, we may try to come as close as possible to such an mathematical authoring environment. For this, we have to ask the user to accept the current limitations of the existing computer programs and follow some well thought guidelines in writing the documents.

The main goal of the mathematical document editing environment we propose is to let the author concentrate on writing. We want to reduce the task of semantically annotating the document the user is working on to a minimum necessary. In order to fruitfully process the finished document we restrict the author to use a certain style for it. Most importantly, the user should:

A. separate text from mathematical formulae; and
B. group the formulae under certain headers (Definitions, Theorems, Propositions, etc.).

When a document respects the A. and B. requirements, a purpose specific document parser is able to

- identify the mathematical content from the rest of the document,
- correctly identify the mathematical knowledge types of the formulae, and
- store the identified knowledge in a form that is usable for other automated activities, e.g. proving.

We envision that advanced tools will take the output of such a dedicated document parser and extract more information from it, like singling out the defined concepts and their properties, generate new knowledge, etc.

3 Environment Description

We believe that there are certain actions that have to be performed from the moment a user decides to write a document with a mathematical content to the point where the document becomes a part of a mathematical knowledge base. We identified three such actions: a) writing the document following some guidelines; b) verifying (parsing) the document; and c) inserting the document into the knowledge base. Which guidelines we mean at point a) will become clear in Subsection 3.1. In the following, we discuss how each of these actions is performed in the environment proposed.

The implementation of our ideas is done in the frame of Mathematica and *Theorema*. The *Theorema* system is designed to assist a mathematician in all of the phases of his or her work (see [3, 5]). It is built on top of the computer algebra system Mathematica [27]. As a mathematical editing environment, Mathematica offers a very good front end support by giving the possibility of combining text, mathematical expressions, graphics, code in the same document, called “notebook”.

Theorema already provides constructors for writing, using, and composing formal, mathematical basic knowledge (**Definition**, **Proposition**, **Theory**, etc.). However, only few attempts wore done in building a base of formal mathematical knowledge in a systematic way, a knowledge base that can be browsed, extended and used for proving or teaching.

The environment we are about to describe is intended to improve this. With this purpose in mind, we have designed a special Mathematica stylesheet and implemented a set of functions for processing the notebooks that make use of it. We will refer to this environment as the "theory development environment".

To start working within the theory development environment the user has to open *Theorema*'s "Library Utilities" palette. This can be done, with the *Theorema* system loaded, by calling **OpenLibraryUtilities**[ ]. The functionality of the buttons on this palette will gradually be explained in the following subsections.

3.1 Writing the Document

To write a document that is to be included in a mathematical knowledge base, the author has to use a certain type of notebook. This will ease the annotation part of the work when typing the document into the computer. The document type we ask to be used employs the stylesheet facilities of Mathematica. A Mathematica stylesheet is a special kind of notebook, defining a set of styles that are allowed to be used in another notebook ([27] section 2.10). As mentioned before, we have defined a special stylesheet that allows annotating the document a user is working on, without his or her explicit awareness. The annotation is done while writing and is not semantic: it only marks cells and groups of cells in the notebook. This stylesheet will facilitate the parsing of the finished document.

The simplest way to get a document with the specific style sheet is to use the 'Open a Template' button on the "Library Utilities" palette. What we obtain is a document like in Figure 3.1. (The figure also presents the "Library Utilities" palette).

The users that are acquainted with the Mathematica front end can also proceed differently, by opening a new notebook and choosing the 'TheoremaTheory' stylesheet for it. We will continue our description with the assumption that the user pressed the suggested button on the palette and has now opened a notebook like in Figure 3.1, which we will call 'the theory notebook' from now on.

The theory notebook is divided into two parts: header and content.

The header part of the theory notebook contains a title and a code, an author, a description, a reference and a dependencies section.

The code cell contains a short string of characters that is associated with the notebook and its content. The user is not compelled to type in a code, though he or she may prefer one that is a kind of compression of the document's title. The reason for this is revealed in the subsections below. When no code is given, one will be generated when the document is verified (subsection 3.2).

The author section is a text cell where the author of the document will put his or her name and the date the file was created.

The description section is reserved for, as its name says, describing in a few sentences what the content of the document is. The author can add here more information about

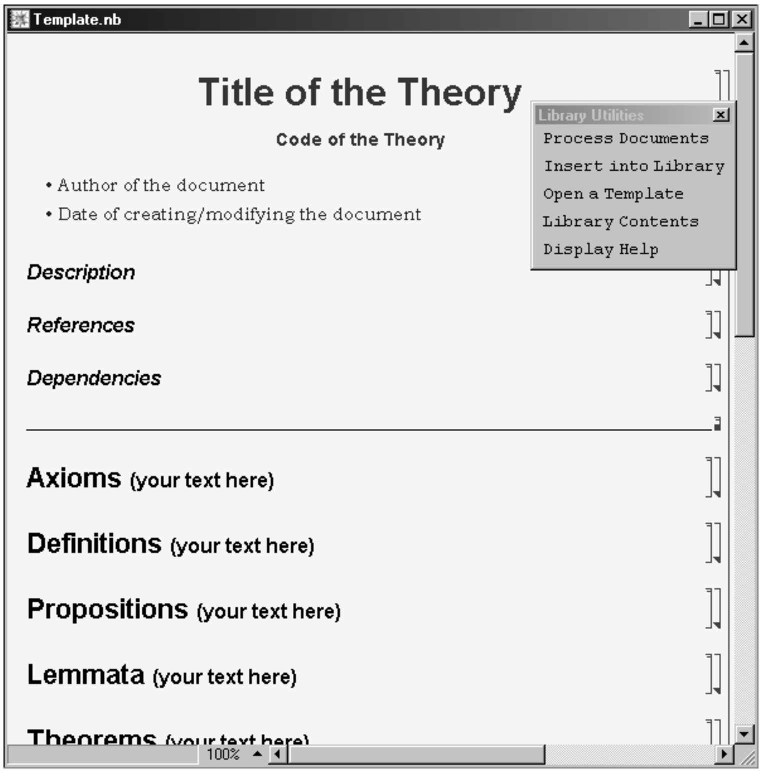


Figure 3.1: Theory notebook template.

the mathematical insights that a human reader may expect to get when reading the document.

In the reference section the author can add pointers to books, web addresses, etc. from where the document content was gathered or where more information can be obtained. The author is free to add other information as well, leaving to his/her common sense that it is relevant for this section.

The dependencies section is giving the author the possibility to specify what other (existing) knowledge is needed in the current document. The author will have to specify, here, the codes of the used theories and the specification of the used parts, if this is the case. (For example, if the author wishes to use the axioms that define the real numbers, which are to be found in an existing document with the code **RealNos**, he or she has to write **Include**[**"RealNos.Axioms"**] in this section)

Only the document title is mandatory to be present in a theory notebook.

The content part of the document is where the actual formulae of the theory are to be typed in. The basic kinds of mathematical knowledge recognized are axioms, definitions, propositions, lemmata, theorems, corollaries, algorithms. The template document provides, for each of them, headings which, based on the style sheet definitions, will mark the formulae underneath them as axioms, definitions, etc. To make it easy to recognize the mathematical expressions we require that the formulae are typed in input cells. This does not put any burden on the authors, since it is the default cell type that will be considered as soon as one starts typing inside a Mathematica notebook.

For example, if a formula is considered to be a proposition it should be written under a heading with the style "Proposition". Though it contributes to clarity, it is not necessary that the word "proposition" appears in the text of the heading. The cell style of the heading has already the information that the formulae that will occur below this header will be propositions. The user can modify the header's text to better reflect the meaning of the formulae underneath it. The author is not restricted to only one section for a knowledge type. (see Figure 3.2)

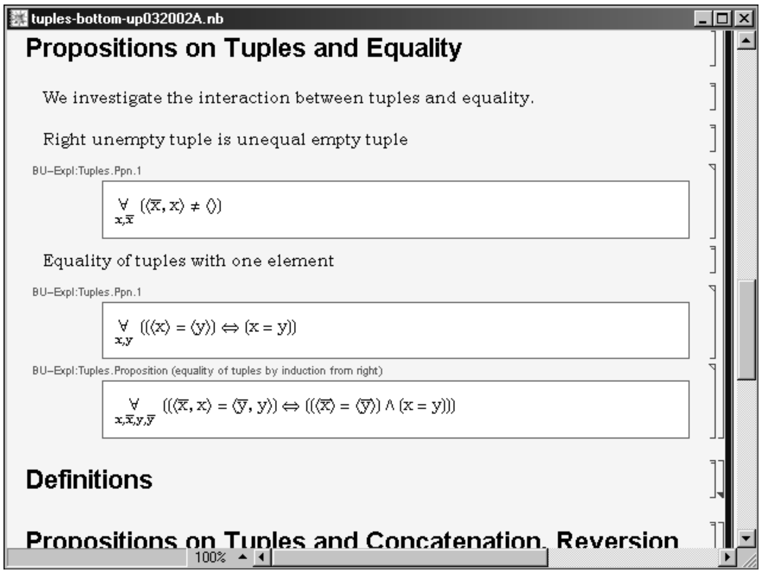


Figure 3.2: Propositions in a theory notebook.

If the author wishes to attach labels to formulae this can be easily done by adding a tag to the cell where they occur. In the document, they will appear in a smaller font just above the formula cell. Tagging cells is a feature of the Mathematica front end (see [27] section 2.11.9). After verifying the document each formula will have a label attached. The labels are used to uniquely identify a formula in a library of theories.

As a final remark to this subsection we mention that the user can add anywhere in the document textual information that helps a human reader understand the presented knowledge.

3.2 Verifying the Document

Starting the verification process is done by pressing the 'Process Documents' button on the "Library Utilities" palette. The stylesheet used for writing the document helps identifying within it the information that is of interest for further processing.

The first step in verifying the document is to check wether the theory notebook has a title and a code. If the title is missing the verifying process stops with an error message. If the code is not present in the theory notebook the verifier will compute one by taking the first letters of the words appearing in the title, and will add it in the notebook.

The verifier will check now the theory code against a list of existing theory codes that it has extracted from the knowledge base. If there is a name clash an error message is returned and the process stops. The author has to correct this problem.

Next, the document verifier will check that the theories and part of theories that are mentioned in the dependencies section are valid and do not lead to circles in the dependencies graph. If there is a loop detected the process stops with an error message and the user has to correct this matter.

Having passed these checks the verifiers will generate and attach labels to the formulae in the file. The generation takes into account the theory code, the knowledge type (axiom, definition, etc) and a numeric counter. This combination will uniquely identify the formula among all the formulae in the knowledge base. When a formula already has a user-given label, the verifier will not generate a label for it, but it will add the theory code in front of it. Figure 3.2 presents a part of a verified theory notebook.

In the end, the verifier will also add a content section in the header part of the document. This section is a compressed image of the content part of the document, having hyperlinks to formulae in it. This is meant to help a human reader to find a formula by just a click on its label.

3.3 Inserting the Verified Document into the Library

The verification process described above can be performed several times. When no errors occurred, the theory document can be inserted into the theory library. This is done by pressing the 'Insert into Library' button on the "Library Utilities" palette.

The procedure will use *Theorema*'s input parsing routines for the mathematical formulae that occur in the document. Each of the formulae will be read, parsed and the proper *Theorema* constructs will be created for it. This is the moment where the annotations made via the style sheet used for editing the document play an important role. A Mathematica package file, that contains the *Theorema* constructs, is created. Loading this package will make available to the *Theorema* system all the formulae that were introduced in the theory notebook. They can be used in the proving process.

At the same time, an entry about the theory notebook is made in a special theory index file. The theory index file keeps a record of each theory notebook that is part of the theory library. This includes information on where the file and its corresponding Mathematica package are stored.

The functionality of the 'Library Contents' button on the "Library Utilities" palette is the following: based on the entries stored in the theory index file, it will dynamically construct and present the user a notebook with a list of theories already existing in the knowledge library. The list has hyperlinks to the notebooks where the theories are introduced.

4 Concluding Remarks and Future Work

We have presented an environment for editing documents, verifying and including them into a mathematical knowledge library. This environment allows the users to concentrate on writing, requiring only that they use a certain style sheet for their documents. A document that uses this style sheet can be automatically processed in order to extract its mathematical content and store it in a format that can be used for browsing, proving, etc. For example, we could apply the tools described in [22] for obtaining derived knowledge.

The theory library that is built using the described environment comprises both the documents written by the authors and the processed files obtained out of them. The reason for this is that a human reader will want to read and inspect the former, while an automated theorem prover will use the latter.

There are features that are missing in our environment and are subject to future work. Among them we mention the plan to improve the routine that extracts the mathematical content from the theory notebook and inserts it into the theory library. For example, automatically identifying the defined symbols in the document should be possible, the user should be allowed to hierarchically organize the formulae in the theory notebook. Also, we did not yet thoroughly consider how searching for notions and concepts can be done best in such a theory library. Another issue is how to manage modifications that the user might perform to the documents that are already included in the theory library.

Bibliography

[1] S. Allen, M. Bickford, R. Constable, R. Eaton, C. Kreitz, L. Lorigo. *FDL: A Prototype Formal Digital Library.* Cornell University, 2002. (http://www.nuprl.org/FDLproject/02cucs-fdl.html)

[2] A. Asperti, B. Buchberger, J.H. Davenport, James Harold (Eds.) Proceedings of the Second International Conference, MKM 2003 Bertinoro, Italy, February 16-18, 2003 Series: Lecture Notes in Computer Science, Vol. 2594, 2003, X, 225 p. Also available online. Softcover ISBN: 3-540-00568-4

[3] B. Buchberger. *Mathematical Knowledge Management Using Theorema.* In [11].

[4] B. Buchberger, G. Gonnet, M. Hazewinkel. *Annals of Mathematics and Artificial Intelligence*, Volume 38, Volume 38, Number 1-3, May 2003. Kluwer Academic Publishers, ISSN 1012-2443.

[5] B. Buchberger. *Theorema: A short introduction.* The Mathematica Journal, 8(2):247–252, 2001.

[6] B. Buchberger. *Algorithm Invention and Verification by Lazy Thinking.* In: D. Petcu, V. Negru, D. Zaharie, T. Jebelean (eds), Proceedings of SYNASC 2003 (Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, October 14, 2003), Mirton Publishing, ISBN 9736611043, pp. 226.

[7] B. Buchberger. *Groebner Rings in THEOREMA: A Case Study in Functors and Categories*, SFB (Special Research Area) "Scientific Computing" Technical Report Nr. 2003 - 46, Johannes Kepler University, Linz, Austria, 2003.

[8] B. Buchberger. *Towards the Automated Synthesis of a Gröbner Bases Algorithm.* RACSAM (Review of the Royal Spanish Academy of Science), to appear, 10 pages.

[9] B. Buchberger. *Computer-Supported Mathematical Theory Exploration: Schemes, Failing Proof Analysis, and Metaprogramming.* Submitted, also available as Technical Report from RISC, Johannes Kepler University, Austria.

[10] B. Buchberger, A. Craciun. *Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema.* In: Fairouz Kamareddine (ed.), Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, Nov. 25, 2003, Electronic Notes on Theoretical Computer Science, volume dedicated to the MKM 03 Symposium, Elsevier, ISBN 044451290X, to appear.

[11] B. Buchberger, O. Caprotti. Editors of the Proceedings of the First International Workshop on Mathematical Knowledge Management: MKM'2001 RISC, A-4232 Schloss Hagenberg, September 24-26, 2001. ISBN 3-902276-01-0.

[12] B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, W. Windsteiger. *The Theorema Project: A Progress Report.* In: Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, August 6–7, 2000, St. Andrews, Scotland, M. Kerber and M. Kohlhase eds.), A.K. Peters, Natick, Massachusetts, pp. 98–113. ISBN 1–56881–145–4.

[13] O. Caprotti, D. Carlisle. *OpenMath and MathML: Semantic Mark Up for Mathematics.* In ACM Crossroads, ACM Press, 1999.

[14] *The Coq proof assistant.* (http://coq.inria.fr/coq-eng.html)

[15] G. Goguadze, A. González Palomo. *Adapting Mainstream Editors for Semantic Authoring of Mathematics.* Presented at the Mathematical Knowledge Management Symposium, November 2003, Heriot-Watt University, Edinbourgh, Scotland.

[16] *IMPS: An Interactive Mathematical Proof System.* Developed at The MITRE Corporation by W. M. Farmer, J. D. Guttman, F. J. Thayer. (http://imps.mcmaster.ca/)

[17] *Isabelle.* Developed at Cambridge University (Larry Paulson) and TU Munich (Tobias Nipkow). (http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html)

[18] M. Kohlhase. *OMDoc: An Infrastructure for OpenMath Content Dictionary Information.* In ACM SIGSAM Bulletin, volume 34, number 2, pages 43-48, 2000.

[19] M. Kohlhase, A. Franke. *MBase: Representing Knowledge and Context for the Integration of Mathematical Software Systems.* Journal of Symbolic Computation 23:4 (2001), pp. 365 – 402.

[20] D.W. Lozier, *NIST Digital Library of Mathematical Functions.* In Annals of Mathematics and Artificial Intelligence, vol. 38, No. 1–3, May 2003. Eds. B. Buchberger, G. Gonnet, M. Hazewinkel. Kluwer Adacemic Publishers, ISSN 1012-2443.

[21] *The Mizar System.* Developed at the University of Warsaw, directed by A. Trybulec. (http://mizar.uwb.edu.pl/system/)

[22] K. Nakagawa, B. Buchberger. *Two Tools for Mathematical Knowledge Management in Theorema*. In [11].

[23] S. Owre, J. Rushby, N. Shankar, D. Stringer-Calvert, *PVS: An Experience Report*, In: Applied Formal Methods—FM-Trends 98. Eds. D. Hutter, W. Stephan, P. Traverso, M. Ullman. LNCS vol. 1641, pp. 338–345. Springer-Verlag, Germany.

[24] F. Piroi. *Tools for Using Automated Provers in Mathematical Theory Exploration*. Ongoing PhD thesis, to be finished in autumn 2004.

[25] S. Rockey. Mathematics Digitization, at Cornell University, Mathematics Library. http://www.library.cornell.edu/math/digitalization.php

[26] W3C Math Home: What is MathML? (http://www.w3.org/Math/)

[27] S. Wolfram. *The Mathematica Book*. Wolfram Media Inc. Champaign, Illinois, USA and Cambridge University Press, 1999.

Integrated Proof Transformation Services

Jürgen Zimmer[1] Andreas Meier[2] Geoff Sutcliffe[3]
Yuan Zhang[3]

[1] School of Informatics, University of Edinburgh, Scotland
jzimmer@inf.ed.ac.uk, http://www.mathweb.org/~jzimmer
[2] DFKI GmbH, Saarbrücken, Germany
ameier@dfki.de, http://www.ags.uni-sb.de/~ameier
[3] Department of Computer Science, University of Miami, USA
{geoff@|yuan@mail.}cs.miami.edu, http://www.cs.miami.edu/~geoff

Abstract

In the last few decades a large variety of mathematical reasoning tools, such as computer algebra systems, automated and interactive theorem provers, decision procedures, etc. have been developed and reached considerable strength. It has become clear that no single system is capable of providing all types of mathematical services, and that systems have to be combined for ambitious mathematical applications. Unfortunately, many mathematical reasoning systems use proprietary input and output formats, and the output in these system-specific formats is often incomprehensible to other components and human users. Transformation tools and data-exchange formats are necessary in order to combine systems and to grant common access to mathematical content. This paper describes the integration of several proof transformation tools in a Java agent architecture, their description in a mathematical service description language, and their combination via a brokering mechanism. The applicability of the approach is demonstrated with an example from group theory.

1 Introduction

In the last few decades a large variety of mathematical reasoning tools, such as computer algebra systems, automated and interactive theorem provers, decision procedures, etc. have been developed and reached considerable strength. Diverse repositories of formalized mathematics have also emerged, e.g., [19]. Despite some successful applications of these systems, none of them have scaled up to a mathematical assistant system providing all kinds of mathematical services. The vision of a powerful mathematical assistant environment that provides supports for most tasks of a working mathematician has recently come into focus, stimulating new projects and international research networks across the disciplinary and systems boundaries. Examples are the European CALCULEMUS [12] (integration of computation and deduction) and MKM [4] (mathematical knowledge management) initiatives, and the American QPQ [14] repository of deductive tools. A main goal in these initiatives is to bring together approaches from different directions. It has become clear that no single system is capable of providing all types of mathematical services, and that systems have to be combined for ambitious mathematical applications. For example, subgoals of one component are commonly delegated to other specialist components, such as automated theorem proving (ATP) systems (e.g., see [27, 6]). Unfortunately, many mathematical reasoning systems use proprietary input and output formats, and the output in these system-specific formats is often incomprehensible to other components and human users. Transformation tools and data-exchange formats are necessary in order to combine systems and to grant common access to mathematical content. This holds, in particular, for the output from ATP systems, because their output often reflects the peculiarities of the internal calculus and proof search procedure.

This paper describes the integration of several proof transformation tools in a Java agent architecture, their description in a mathematical service description language, and their combination via a brokering mechanism. The proof transformation tools provide the following functionality: 1) The Otterfier system translates arbitrary first-order resolution proofs into resolution proofs whose inference steps use only the inference rules of the Otter system [15]. 2) The Tramp system [16] translates problems in full first order form (FOF) logic to equisatisfiable clause normal form (CNF), and translates resolution proofs into proofs in the Natural Deduction (ND) calculus at the assertion level. 3) The *P.rex* system [10] translates ND proofs into natural language. Ideas from Semantic Web research are being adopted to express the functionality of these tools in the service description language MSDL [3]. A brokering mechanism is then used to combine the proof transformation services, to provide customized compound services that are capable of answering queries from other reasoning systems or human users. For example, given a conjecture in classical first-order predicate logic, a proof assistant system could ask for a resolution or ND proof of the conjecture. A human user of the proof assistant may extend the query to request a natural language version of the proof.

The transformation tools and their combination depends heavily on the newly emerging TSTP data-exchange format for problems and proofs [23]. Specifically useful for this work, TSTP defines a syntax for problems in FOF and in CNF. and a format for resolution style derivations. A refutation in TSTP contains *initial clauses*, i.e. the clauses given to an ATP system or produced by its CNF generator, and the *derived clauses* together with the inference rules used to derive them.

This paper is structured as follows: First the various systems involved are introduced in section 2. Section 3 presents the services offered by the systems, and the brokering mechanism that combines these services. The integration approach is explained with a sample application in section 4. Finally, section 5 concludes with some discussion of related and future work.

2 The Systems Involved

2.1 Automated Theorem Proving Systems

This work makes use of the ATP systems Otter and EP. Otter [15] is designed to prove theorems stated in first-order logic with equality. Otter's inference rules are based on resolution and paramodulation, and it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing and restricting searches for proofs. Otter is particularly interesting for our application because the inference rules used in Otter refutation proofs are quite limited, and Otter proofs can therefore readily be used by the proof transformation system Tramp (see section 2.3). A modified version of Otter is also used in Otterfier to transform resolution proofs (see section 2.2).

EP is an equational theorem prover, combining the E system [21] with a proof analysis tool for extracting the required inference steps. The calculus of EP combines superposition (with selection of negative literals) and rewriting. No special rules for non-equational literals have been implemented, i.e., resolution is simulated via paramodulation and equality resolution. On the one hand, EP is typically much stronger than Otter when proving theorems in automatic mode (cf. CADE system competitions 15 and 16). On the other hand, EP uses rather complex inference rules which makes its proofs hard to process with other systems (e.g., Tramp - see section 2.3).

2.2 Otterfier - A CNF Derivation Transformer

The derivations (typically refutations) output by contemporary CNF based ATP systems are built from inference steps, which have one or more parent clauses and one resultant inferred clause. The inference rules that create the steps vary depending on the ATP system, ranging from simple binary resolution through to complex rules such as superposition [2]. In almost all cases the inferred clauses are logical consequences of their parent clauses, the most common exception being clauses resulting from the various forms of splitting that have been implemented in ATP systems such as Vampire [18], E, and SPASS [24]. While a wider range and complexity of inference rules typically improves the performance of ATP systems, it is impractical to require proof postprocessing tools to be able to process inference steps created by all the various rules (and new ones that may be invented in the future). It is therefore desirable to be able to transform derivations so that each inference step uses one of a limited selection of inference rules that are amenable to a range of postprocessing operations. The Otterfier system is a transformation tool that transforms a *source derivation* containing *source inference steps* of logical consequence, to a derivation whose inference steps use only selected inference rules available in Otter. The transformation is independent of the inference rules used in the source inference steps, relying only on the inferred clauses being logical consequences of their parent clauses.

Otterfier uses a modified version of Otter. The standard Otter system includes the *hints* strategy. Hints are normally used as a heuristic for guiding the search, in particular, in selecting the given clauses and in deciding whether to keep derived clauses. The fast version of the hints strategy, called *hints2* in Otter, allows the user to specify a set of clauses against which newly inferred clauses are tested for subsumption. For Otterfier the hints strategy has been modified so that when a newly inferred clause is equal to or subsumes a hint, the search is halted and the derivation of the newly inferred clause is output. This modified strategy is called the *target* strategy.

The basic mechanism of Otterfier is to place the parents of a source inference step into Otter's set-of-support list, and the inferred clause into Otter's hint list. The inferred source clause is called the *target clause* in this context. Otter is then run with a complete selection of inference rules, e.g., binary resolution, factoring, and paramodulation. As Otter derives the logical consequences of the parent clauses, the target strategy checks each logical consequence against the target clause in the hints list. When the target clause is derived or subsumed, the derivation output by Otter provides a transformed version of the source inference step, using only Otter's inference rules. A source derivation is transformed by performing this transformation on each source inference step, and the combined transformed steps form a complete transformed derivation.

The search strategy of the modified Otter is the default strategy of the normal Otter, i.e., aimed at finding a refutation of the input clauses. If the parent clauses of a source inference step are satisfiable, as are the parents of a source inference step in most cases, then no refutation can be found (see below for the case when the parent clauses are unsatisfiable). In this situation Otter derives clauses with a focus on clauses with lower symbol count. As the number of clauses with a given symbol count is finite, Otter derives longer and longer clauses as its search progresses, eventually deriving clauses whose length is that of the target clause. By that stage the target clause can be derived or subsumed. While Otter can be configured to be refutation complete, it is not known to be deduction complete, i.e., it is possible that the target clause may never be derived or subsumed. In practice this possibility has not yet been encountered, and if it does occur it will merely be a cause of incompleteness of the transformation process.

There are several special outcomes from the target strategy that allow Otterfier to optimize the transformed derivation. First, if the clause derived by Otter subsumes the target clause (rather than being only equal to the target clause), Otter's derived clause replaces the corresponding parent clause in subsequent source inference steps. This maintains the coherency of the transformed derivation. Second, the clause derived by Otter might subsume the inferred clause of a subsequent source inference step of which the current target clause is a parent or ancestor. In this situation the subsequent inference step is removed from the source derivation, and Otter's derived clause replaces the inferred clause of the subsequent inference step. Third, if the source derivation is a refutation, i.e., ends with an empty clause, and Otter derives the empty clause while searching for a non-empty target clause, then a transformed refutation is created, consisting of only the transformed inference steps that end at Otter's derived empty clause. Both the second and third special cases allow Otterfier to produce a transformed derivation that is, in some sense, shorter than the source derivation. The discovery of such derivations is of interest in it's own right [25].

2.3 Tramp - A Natural Deduction Proof Generator

The Tramp system [16] can transform the output of several automated theorem provers into natural deduction proofs at the assertion level [13]. The assertion level allows for human-oriented macro-steps justified by the application of theorems, lemmas, or definitions, which are collectively called *assertions*. For instance, the assertion level step

$$\frac{F{\subset}G \qquad c{\in}F}{c{\in}G}{\subset}\mathrm{DEF}$$

derives the conclusion $c \in G$ by an application of the subset definition $\subset$DEF — formalized by $\forall S_1.\forall S_2.(S_1 \subset S_2 \Leftrightarrow \forall x.(x \in S_1 \Rightarrow x \in S_2))$— from the premises $c \in F$

and $F \subset G$. A corresponding base ND proof, including the expansion of the subset definition, consists of a lengthy sequence of ND steps.

Tramp consists of a set of transformation procedures. First, there are transformation procedures that take a proof output from an ATP system and transform it into an internal resolution proof object. In its current distribution Tramp is able to process the output of the ATP systems SPASS, Bliksem, Otter, Waldmeister, ProTeIn, and EQP. At the heart of Tramp is a transformation procedure that creates an ND proof at the assertion level. Finally, the resulting ND proof at the assertion level can be further processed. In particular, each assertion application can be expanded such that the resulting proof is a pure ND proof without assertion application steps. Tramp can output its proofs in LaTeX format as well as in the languages $\mathcal{POST}$ and TWEGA (cf. section 2.4).

The original Tramp system could take only one input: a description of a FOF problem in $\mathcal{POST}$ syntax. Tramp then translated the FOF problem into its equivalent CNF and called one of the supported ATP systems. The result of the ATP system was then converted into Tramp's internal format and an ND proof for the original conjecture was created. The reason for the CNF creation within Tramp is that, in order to create a ND proof, Tramp has to establish a connection between the literals of a resolution proof and the corresponding literal sub-formulae in the original FOF problem.

In order to employ Tramp in the integrated proof transformation system, it was necessary to extend Tramp in three ways: (1) A new input module for TSTP resolution proofs was developed. Currently, this TSTP module exists in parallel with the transformation procedures for the ATP systems supported by Tramp. However, TSTP will eventually become the only necessary input format of Tramp. (2) Tramp now accepts two inputs: the FOF description of the original conjecture and a TSTP resolution proof of the conjecture. Tramp tries to map the literals of the initial clauses in the resolution proof to the corresponding literal sub-formulae of the original first-order formulae. However, Tramp can compute this mapping only if the initial clauses of the resolution proof comply with Tramp's clause normal form (CNF) algorithm (see also section 3). If this is the case Tramp produces an ND proof of the FOF problem. (3) A FOF problem generating procedure was added. This is activated if no FOF problem is provided as input, or if Tramp cannot compute the relationship between the literals of the resolution proof and the literal sub-formulae of the FOF problem. The procedure computes a FOF problem description that corresponds to the initial clauses in a given resolution proof, by creating a disjunction of the literals in each clause and universally quantifying all variables. Since the transformation procedure cannot distinguish between Skolem functions and other functions, it interprets every function symbol as a function of the input signature, and does not create any existential quantifications.

The extensions are important for the use of Tramp in the integrated transformation system: The first allows Tramp to work on TSTP proofs and to be coupled with other reasoning systems, such as Otterfier. The second makes Tramp independent of the ATP system used. The third reduces the necessary input and enables a broader application of Tramp.

2.4 *P.rex* - A Natural Language Proof Generator

P.rex [10] is an interactive natural language proof explanation system for machine-generated proofs. It adapts its explanations to the user and can interact with the user by questions and requests [9]. In the context of the integrated transformation system the full functionality of *P.rex* is not exploited – it is used only to obtain a single natural language explanation of a proof.

P.rex is based on a logical framework and uses the formal language TWEGA for inputting proofs. Since Tramp can output TWEGA format, Tramp's ND proofs can be further processed by *P.rex*. A proof is handled by *P.rex* in two steps: First, a dialog plan is created, and this is then passed to a presentation component that creates a natural language presentation [10]. *P.rex* can provide its natural language explanation in ASCII, LaTeX, and a markup language similar to HTML.

The (quality of the) output of *P.rex* depends on the availability of linguistic knowledge. Linguistic knowledge is stored in a database that is structured in theories. In order to make use of the linguistic knowledge the theory to which the problem belongs has to be provided as the second input to *P.rex*. This feature is not yet used in the integrated transformation system.

3 Combination of Transformation Services

A framework for describing the capabilities of deduction systems in a formal service description language was introduced in [26]. These "semantic" descriptions can be used for service discovery and brokering, i.e., the search for services suitable for tackling a given problem. A *broker* can also use the service descriptions to dynamically combine systems to solve a given problem. Human users or reasoning systems can simply send queries to a broker and wait for a result. Following this idea, some of the functionality of the systems described in section 2 have been captured in the service description language MSDL [3]. This section briefly describes the ontology underlying the service descriptions, the descriptions themselves, and the brokering mechanism.

3.1 An Ontology for Service Descriptions

Semantic descriptions of services offered by deductive components depend on a commonly agreed ontology. We are currently developing such an ontology in the Web Ontology Language (OWL) [8]. Only a small fragment of the ontology, as needed to describe the services offered by the systems of section 2, is described here - to increase readability most properties are not shown. Figure 3.1 shows the "is-a" (subclass) relationship between some concepts of the ontology as solid lines with arrows. Properties and their cardinality restrictions are denoted with dashed lines. Individuals (instances) are connected to their concepts by dotted lines.

The concept Proving-Problem is crucial for this paper. In addition to the axioms and the conjecture that ought to be proven, a Proving-Problem contains information such as the logic in which the problem is defined and resource limits. A Proving-Problem can be further specialized to a FO-Proving-Problem which is formulated in first-order predicate logic, and then to TSTP problems in CNF or FOF syntax [23].

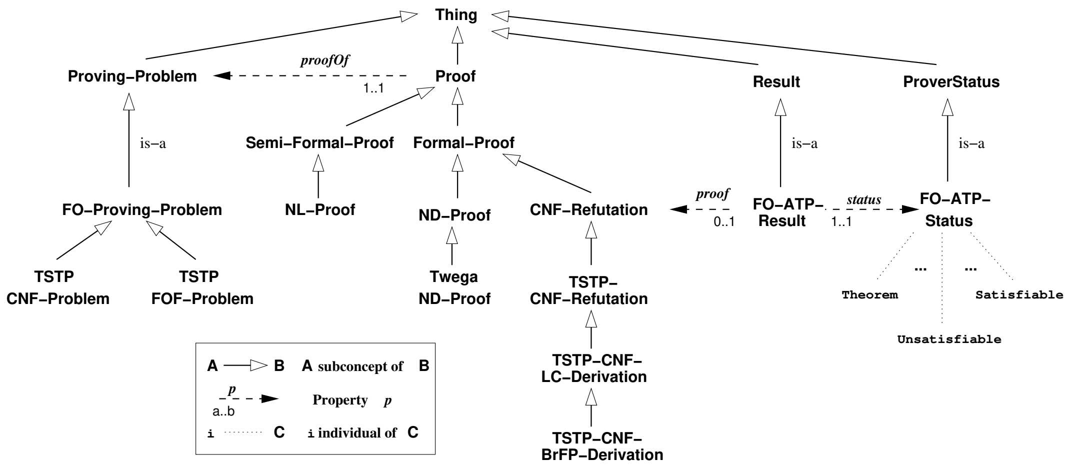


Figure 3.1: A fragment of the ontology for deduction services

Another crucial concept for this paper is FO-ATP-Result, which denotes results of first-order ATP systems. The *status* of a FO-ATP-Result always contains one of the valid statuses of first-order ATP systems as described in [23]. This status describes unambiguously what an ATP system has established about the given problem, e.g., the status `Unsatisfiable` means that the system has established that the given set of clauses is unsatisfiable. A FO-ATP-Result can also have a *proof* property, which can contain at most one proof of a given conjecture (the range of the property *proofOf* is the concept Proof). Certain domain-specific consistency rules apply such as, for instance, that a FO-ATP-Result must not contain a *proof* if its *status* is `Unknown`.

The concept of Proofs subsumes Semi-Formal-Proofs, e.g., natural language proofs, and Formal-Proofs in different logical calculi, e.g., ND or resolution calculus. Twega-ND-Proofs are special ND-Proofs in the TWEGA language. TSTP-CNF-Refutations are refutation proofs in TSTP format. A TSTP-CNF-LC-Refutation employs only inference rules that produce logical consequences. The latter can be further restricted to a TSTP-CNF-BrFP-Refutation which employs only binary resolution, paramodulation, and factoring.

3.2 Proof Transformation Services

The projects *MathBroker* [20] and MONET [5] have developed the Mathematical Service Description Language (MSDL) [3] to semantically describe reasoning services on the Semantic Web. Although MSDL aims at describing all kinds of mathematical services, the two projects have, so far, investigated only the description of symbolic and numeric computation services. We are using our expertise in deduction systems to extend the use of MSDL to deduction services.

An MSDL document describes many different facets of a service. In what follows we present only the facet important for this paper, namely the abstract mathematical problem that can be solved.[1] As an example a generic first-order theorem proving

[1]Those parts of the MSDL description needed for further classification of the service and for the

service, GenericATP, is presented. GenericATP is provided by an ATP system such as EP. To increase readability, the service description is presented in a table rather than in the XML syntax of MSDL.

Service: GenericATP	
input parameters:	*problem*::TSTP-CNF-Problem
output parameters:	*result*::FO-ATP-Result
pre-conditions:	$\top$
post-conditions:	*proof(?result, ?proof)* $\Rightarrow$ *type(?proof, TSTP-CNF-Refutation)*

GenericATP has one input, a clause normal form of a conjecture in TSTP format (TSTP-CNF-Problem), and one output, a FO-ATP-Result. A '::' is used to separate the name of a parameter (e.g., *result*) from the RDF type information in the MSDL description (e.g., FO-ATP-Result). It is important to note that GenericATP always delivers a FO-ATP-Result after the given time resource[2] is used up. However, the result might not contain a proof. The **pre-conditions** of an MSDL service state service-specific conditions the input parameters have to fulfill. The **post-conditions** can give further information about the output parameters and can relate input and output parameters. At the moment, pre- and post-conditions may contain RDF triples on concept properties, conjunctions of triples, and Horn clauses. GenericATP has no pre-conditions - they are simply set to $\top$. Its post-conditions say that if the result contains a proof then it is a CNF refutation proof in TSTP format.

The following paragraphs describe the services provided by the systems introduced in section 2.

The Otterfier Service. OtterfierService takes the result of any ATP system and tries to transform the refutation proof in it, if existent, into a TSTP-CNF-BrFP-Refutation which contains only applications of binary resolution, factoring and paramodulation (the BrFP calculus). The fact that Otterfier is based on a modified Otter justifies that the service returns a FO-ATP-Result:

Service: OtterfierService	
input parameters:	*oldResult*::FO-ATP-Result
output parameters:	*newResult*::FO-ATP-Result
pre-conditions:	$\top$
post-conditions:	*proof(oldResult, ?oldProof)* $\wedge$ *proof(newResult, ?newProof)* $\wedge$ *type(?newProof, TSTP-CNF-BrFP-Refutation)* $\wedge$ *altProof(?newProof, ?oldProof)*

The post-conditions of OtterfierService express that the newly generated proof is a TSTP-CNF-BrFP-Refutation, and is an alternative refutation proof of the same conjecture.[3]

service grounding, i.e. low-level details about how to invoke the service, are omitted.

[2]The time resource is a property of the concept Proving-Problem and, hence, also of the TSTP-CNF-Problem input of the service.

[3]As a side effect, this equivalence provides a semantic verification of the original refutation.

Services offered by Tramp. Since GenericATP accepts only CNF problems, a clause normalization service is needed for problems in FOF format. This is provided by Tramp, which ensures that the resulting CNF is compatible with Tramp's ND proof generation routines. Thus, the first service of Tramp transforms a FOF problem into a CNF:

Service: ClauseNormalizer	
input parameters:	*fofProblem*::TSTP-FOF-Problem
output parameters:	*cnfProblem*::TSTP-CNF-Problem
pre-conditions:	$\top$
post-conditions:	*sat-equiv*(fofProblem, cnfProblem)

The fact that the new CNF problem is satisfiability-equivalent to the initial FOF problem is expressed in the post-conditions of ClauseNormalizer.

The second service of Tramp expects two inputs: a FOF problem in TSTP format, and the result of an ATP system. Furthermore, the result of the ATP system should contain a TSTP-CNF-BrFP-Refutation proof. If Tramp can match the literals in the refutation's initial clauses with literal sub-formulae in the FOF problem, then the service returns an ND proof for the FOF problem. The service fails if the initial clauses of the refutation proof are not compatible with Tramp's CNF generator. It is very difficult though to express this constraint in the pre-conditions of the service. It is therefore kept implicit.

Service: NDforFOF	
input parameters:	*fofProblem*::TSTP-FOF-Problem *atpResult*::FO-ATP-Result
output parameters:	*ndProof*::Twega-ND-Proof
pre-conditions:	*proof(atpResult, ?proof)* $\wedge$ *type(?proof, TSTP-CNF-BrFP-Refutation)*
post-conditions:	*proofOf*(*ndProof*, *fofProblem*)

Tramp's third service takes the result of an ATP system, which should contain a TSTP-CNF-BrFP-Refutation proof. Tramp internally creates a first-order problem from the initial clauses in the proof, and transforms the refutation proof into a ND proof for this problem. The service returns the ND proof as well as the newly generated FOF problem:

Service: NDforCNF	
input parameters:	*atpResult*::FO-ATP-Result
output parameters:	*ndProof*::Twega-ND-Proof *fofProblem*::TSTP-FOF-Problem
pre-conditions:	*proof(atpResult, ?proof)* $\wedge$ *type(?proof, TSTP-CNF-BrFP-Refutation)*
post-conditions:	*proofOf*(*ndProof*, *fofProblem*)

The P.rex Service. As mentioned above, only some of the functionality of *P.rex* is used, to produce a natural language proof in LaTeX. The *P.rex* service gets a TWEGA proof in the ND calculus and provides a natural language proof in LaTeX format:

Service: PrexND2NL	
input parameters:	*ndProof*::Twega-ND-Proof
output parameters:	*nlProof*::NL-Proof
pre-conditions:	$\top$
post-conditions:	*proofOf*(*ndProof*, *?p*) $\wedge$ *informalProofOf*(*nlProof*, *?p*)

The post-conditions state that the result is an informal natural language proof of the conjecture proved by the ND proof.

3.3 Brokering of Proof Transformation Services

Using the service descriptions introduced above the broker can combine them to obtain customized compound services. The broker currently translates MSDL service descriptions into plan operators. Queries are translated into an initial state of a modified partial order planner. The planner uses the plan operators to come up with a suitable combination of services that might answer the given query. These partially ordered plans are linearized and translated into an execution protocol. For instance, if a proof assistance system comes up with an open conjecture in first-order logic, it can encode it in TSTP FOF format and ask the broker to deliver an ND proof for the conjecture. Assuming some ATP service (GenericATP), the OtterfierService, and the two Tramp services are available, the broker can then simply combine the four services. Figure 3.2 shows the resulting combination. Some post-conditions of the services are also shown with dashed arrows.

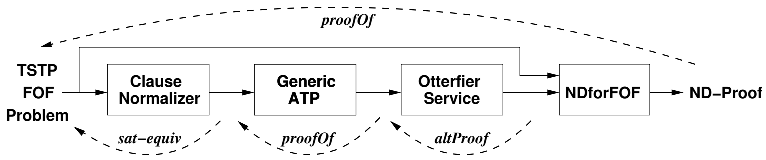


Figure 3.2: A combination of four services

4 Example Scenario

In this section some possible combinations of the transformation services are demonstrated with an example. Imagine that there are three users of our system, Peter, Susan and Mary. They all want to prove the following problem from group theory:[4]

> *Let F be a group and U a subset of F. Moreover, for U the so-called subgroup-criterion holds: if X, Y belong to U, then $X \circ Y^{-1}$ belongs to U (where $\circ$ is the operation and $^{-1}$ is the inverse function of the group F). Then, U is closed wrt. to the inverse function of F, i.e., U contains X^{-1} whenever it contains X.*

All three users managed to formalize the problem as FOF formulae. They could then use more advanced tools like the Java OpenMath editor (JOME) [7] to input the

[4]Problem GRP006 in the TPTP library [22], although a slightly different formalization is used here.

formulas typing a Maple like syntax. The resulting OpenMath formulas could then be translated automatically into TSTP format. The TSTP problem descriptions could, among others, contain the following formulae:

```
fof(subset,axiom, ( ! [SG,G] :
          ( subset(SG,G)
            <=> ! [X] : ( member(X,SG) => member(X,G) )))).
fof(inverse,axiom, ( ! [G,X] :
    ( ( group(G) & member(X,G) )
        => ( member(inverse(G,X),G)
             & equal(multiply(G,X,inverse(G,X)),identity(G))
             & equal(multiply(G,inverse(G,X),X),identity(G)))))).
               ...
fof(f_group,hypothesis, ( group(f) )).
fof(u_subset,hypothesis, ( subset(u,f) )).
fof(subgroupcriterion,hypothesis, ( ! [X,Y] :
            ( (member(X,u) & member(Y,u))
              => member (multiply(f,X,inverse(f,Y)),u) ))).
fof(prove_this,conjecture, ( ! [V] :
            ( member(V,u)
              => member(inverse(f,V),u) ))).
```

In this formalization the function *inverse* as well as the operation *multiply* are parameterized w.r.t. the group structure to which they belong. *identity* is a function whose value is the unit element of the group.

Proving the Theorem

Peter, the first user of the system, is an expert in automated deduction and can understand resolution proofs. Thus he sends the FOF formalization of the problem to the broker asking for a CNF Refutation in TSTP format. The broker simply combines Tramp's ClauseNormalizer service (to create the corresponding CNF problem) and the GenericATP service offered by the EP system. The resulting refutation proof in TSTP format contains, among others, the following clauses:

```
cnf(1,axiom,( member(X1,X2) | ~member(X1,X3) | ~subset(X3,X2))).
  ...
cnf(10,axiom,( equal(multiply(X1,X2,inverse(X1,X2)),identity(X1))
             | ~group(X1) | ~member(X2,X1))).
cnf(12,axiom,(group(f))).
cnf(13,axiom,(subset(u,f))).
cnf(14,axiom,( member(multiply(f,X1,inverse(f,X2)),u)
             | ~member(X1,u) | ~member(X2,u))).
cnf(15,conjecture,(member(sk2,u))).
  ...
cnf(31,derived,( member(identity(f),u)
               | ~member(X1,u) | ~group(f) | ~member(X1,f)),
        inference(pm,[status(thm)],[14,10,theory(equality)])).
  ...
cnf(273,derived,(~member(sk2,f)),
    inference(rw,[status(thm)],[270,15,theory(equality)])).
```

```
cnf(274,derived,(false),
    inference(rw,[status(thm)],[273,51,theory(equality)])).
```

The clauses marked as `axiom` and `conjecture` are derived by clause normalization from the FOF problem formalization, e.g., clause 1 is derived from axiom `subset`. Derived clauses, such as clauses 31, 273, and 274, have a justification listing the inference rule used to derive the clause and the parent clause names. For example, clause 273 was derived from clauses 270 and 15 by EP's `rw` rule, which implicitly uses axioms of equality.

Generating an ND Proof.
Susan, the second user has some knowledge of first-order logic, but knows nothing about clauses and the resolution calculus. She also submits the problem to the broker but asks for a ND proof instead. The broker comes up with the combination of services shown in Figure 3.2. The OtterfierService transforms the refutation shown above into a new refutation containing, among others, the following clauses:

```
cnf(1,initial,(member(A,B)|~member(A,C)|~subset(C,B))).
  ...
cnf(31,derived,(member(identity(f),u)|~member(A,u)|~group(f)|~member(A,f)),
    inference(factor_simp,[status(thm)],[
        inference(para_from,[status(thm)],[10,14,theory(equality)])])).
  ...
cnf(273,derived,(~member(sk2,f)),inference(binary,[status(thm)],[270,15])).
cnf(274,derived,(false),inference(binary,[status(thm)],[51,273])).
```

Note how Otterfier transformed EP's single `pm` inference step (from clauses 10 and 14 to clause 31) to two inferences using Otter's `para_from` and `factor_simp` inference rules. In some cases a fully separate TSTP inference step containing the intermediate inferred formula (rather than a single TSTP step containing two Otter inferences as here) may be generated by Otterfier. Altogether, the resolution proof output by Otterfier consists of 23 clauses, 9 initial and 14 derived.

Finally, the NDforFOF service is invoked with the FOF problem formalization of the problem and the FO-ATP-Result of the OtterfierService. The underlying Tramp creates an ND proof in linearized style as introduced in [1]. The lines of the proof are of the form $L.\ \Delta\vdash F\ (\mathcal{R})$, where L is a unique label, $\Delta\vdash F$ a sequent denoting that the formula F can be derived from the set of hypotheses Δ, and $(\mathcal{R})$ is a justification expressing how the line was derived. Tramp starts with an initial open ND proof that consists of the axioms of the FOF problem and the conjecture. Each axiom becomes an initial hypothesis (justified by *Hyp*), the conjecture is the initial goal (justified by *Open*). The initial ND proof for the problem is as follows:[5]

[5]Variables are now written with lower case letters, and constants are capitalized.

$\subset DEF.$	$\subset DEF$	$\vdash \forall s', s_\blacksquare subset(s', s) \Leftrightarrow \forall x_\blacksquare (member(x, s') \Rightarrow member(x, s))$	(Hyp)
$UnitAx.$	$UnitAx$	$\vdash \forall g_\blacksquare group(g) \Rightarrow (member(identity(g), g) \wedge \forall x_\blacksquare (member(x, g) \Rightarrow (multiply(g, x, identity(g)) = x \wedge multiply(g, identity(g), x) = x)))$	(Hyp)
$InvAx.$	$InvAx$	$\vdash \forall g, x_\blacksquare (group(g) \wedge member(x, g)) \Rightarrow (member(inverse(g, x), g) \wedge multiply(g, x, inverse(g, x)) = identity(g) \wedge multiply(g, inverse(g, x), x) = identity(g))$	(Hyp)
$Criterion.$	$Criterion$	$\vdash \forall x, y_\blacksquare (member(x, U) \wedge member(y, U)) \Rightarrow member(multiply(F, x, inverse(F, y)), U)$	(Hyp)
$FGroup.$	$FGroup$	$\vdash group(F)$	(Hyp)
$U \subset.$	$U \subset$	$\vdash subset(U, F)$	(Hyp)
$Conj.$	$\mathcal{H}$	$\vdash \forall x_\blacksquare member(x, U) \Rightarrow member(inverse(F, x), U)$	$(Open)$

$$\mathcal{H} = \subset DEF, UnitAx, InvAx, Criterion, FGroup, U \subset$$

During the transformation of the resolution proof Tramp adds justification steps and nodes to the ND proof until all nodes are justified. The complete ND-proof at assertion level created by Tramp is (only the new lines and justifications):

$L2.$	$L2$	$\vdash member(C, U)$	(Hyp)
$L4.$	$\mathcal{H}_1$	$\vdash member(multiply(F, C, inverse(F, C)), U)$	$(Criterion\ \ L2)$
$L5.$	$\mathcal{H}_2$	$\vdash member(C, F)$	$(\subset DEF\ \ U \subset\ \ L2)$
$L6.$	$\mathcal{H}_3$	$\vdash multiply(F, C, inverse(F, C)) = identity(F)$	$(InvAx\ \ FGroup\ \ L5)$
$L7.$	$\mathcal{H}_4$	$\vdash member(identity(F), U)$	$(=Subst\ \ L4\ \ L6)$
$L8.$	$\mathcal{H}_4$	$\vdash member(multiply(F, identity(F), inverse(F, C)), U)$	$(Criterion\ \ L7\ \ L2)$
$L9.$	$\mathcal{H}_3$	$\vdash member(inverse(F, C), F)$	$(InvAx\ \ FGroup\ \ L5)$
$L10.$	$\mathcal{H}_5$	$\vdash multiply(F, identity(F), inverse(F, C)) = inverse(F, C)$	$(UnitAx\ \ FGroup\ \ L9)$
$L3.$	$\mathcal{H}, L2$	$\vdash member(inverse(F, C), U)$	$(=Subst\ \ L8\ \ L10)$
$L1.$	$\mathcal{H}$	$\vdash member(C, U) \Rightarrow member(inverse(F, C), U)$	$(\Rightarrow I\ \ L3)$
$Conj.$	$\mathcal{H}$	$\vdash \forall x_\blacksquare member(x, U) \Rightarrow member(inverse(F, x), U)$	$(\forall I\ \ L1)$

$$\begin{aligned}
\mathcal{H} &= \subset DEF, UnitAx, InvAx, Criterion, FGroup, U \subset \\
\mathcal{H}_1 &= Criterion, L2 \\
\mathcal{H}_2 &= \subset DEF, U \subset, L2 \\
\mathcal{H}_3 &= InvAx, FGroup, \subset DEF, U \subset, L2 \\
\mathcal{H}_4 &= Criterion, InvAx, FGroup, \subset DEF, U \subset, L2 \\
\mathcal{H}_5 &= UnitAx, InvAx, FGroup, \subset DEF, U \subset, L2
\end{aligned}$$

Here the justifications $\forall I$ and $\Rightarrow I$ of the nodes $L1$ and $Conj$ are the basic ND rules introduction of universal quantification and introduction of implication. $=Subst$, which is used in the justifications of node $L3$ and $L7$, is the ND rule for equality substitution. All other justifications are assertion applications. For instance, the justification $(\subset DEF\ \ U \subset\ \ L2)$ of node $L5$ is the application of assertion $\subset DEF$ to the premises $U \subset$ and $L2$. Altogether the resulting ND proof at the assertion level consists of 17 nodes and 6 assertion steps. When all complex steps are expanded, then the resulting basic level ND proof consists of 54 nodes.

The main – clearly comprehensible – steps in the direct ND proof are: First, assume that an arbitrary C is in U (in $L2$). Then, use the subgroup-criterion to derive that the identity of F is in U (in $L7$). Finally, use again the subgroup-criterion to derive that the inverse of C is in U.

From ND to NL.

Mary knows only mathematical proofs as they are presented in textbooks. She asks the broker to return a natural language proof for the problem. The broker therefore extends the service sequence in Figure 3.2 with the PrexND2NL service. The *P.rex* system underlying this service can access basic linguistic knowledge about first-order logic connectives but doesn't have any knowledge about the particular domain or the problem.

From the ND proof at the assertion level, *P.rex* creates a natural language proof. The relevant parts of the Postscript version of the proof are shown in Figure 4.3. In the verbalization of assertion applications, this natural language proof refers to the axioms of the ND proof, e.g., $\subset DEF$ and $Criterion$. In the complete verbalization, which we skip here, these axioms are introduced and verbalized as well.

> [...] Let $member(C, U)$. Then $member(C, F)$ because $subset(U, F)$ by $\subset DEF$. Thus $member(inverse(F, C), F)$ because $group(F)$ by $InvAx$. That implies that $multiply(F, identity(F), inverse(F, C)) = inverse(F, C)$ by $UnitAx$ since $group(F)$. That implies that $member(multiply(F, C, inverse(F, C)), U)$ by $Criterion$. That leads to $multiply(F, C, inverse(F, C)) = identity(F)$ by $InvAx$ because $group(F)$. That implies that $member(identity(F), U)$. Therefore $member(multiply(F, identity(F), inverse(F, C)), U)$ by $Criterion$. That implies that $member(inverse(F, C), U)$. Therefore $member(C, U)$ implies that $member(inverse(F, C), U)$. That implies that $member(x, U)$ implies that $member(inverse(F, x), U)$ for all x.

Figure 4.3: Fragment of the *P.rex* proof verbalization with basic linguistic knowledge

Mary still has some problems understanding the proof because *P.rex* was not given any linguistic knowledge about the domain. Simple facts, such as the expression $subset(t, t')$, should be written as $t \subset t'$. Futhermore, $member(x, s)$ should be written as $x \in s$. This would considerably improve the readability of the proof.

It is important to note that our brokering mechanism acts as a black box and Mary does, for instance, not have to work with theorem provers on the level of clauses and resolution proofs.

Figure 4.4 shows the length of the four different proofs as well as the time for finding/transforming the proofs involved. The run times were measured on a Pentium IV 2GHz machine. The time of the Otterfier service is CPU time while all other times are wall clock times. Both resolution proofs contain 9 initial clauses, all other clauses are derived. The ND proof contains 6 assertion level steps and 4 ND calculus steps. *P.rex* does not use any domain-specific linguistic knowledge while translating the ND proof into natural language.

Proof	**Proof Length**	**Time (secs)**	**Format/Calculus**
EP proof	19 clauses	0.031	EP
Otterfier proof	30 clauses	15	BrFP
Tramp proof	10 ND steps	4	ND
P.rex proof	10 sentences	54	NL

Figure 4.4: Lengths and proving times for the different proofs

5 Conclusion and Future Work

Ongoing work on integrated proof transformation services and their dynamic combination has been presented. By using the TSTP data-exchange format, and by defining the notion of a resolution proof in a restricted calculus, it has been possible to combine several independently developed systems that could not previously interact with each other. The extension and integration of the systems into a Java framework has been completed, and a prototypical version of the broker has been implemented. Efforts are underway to make the services available as web services. An execution framework for the service sequences planned by the broker will be implemented in the near future. The outcome is an integrated service that can support mathematical reasoning from problem specification in first-order logic through to proof presentation in natural language. Our system absolves the users from the need to know details of system specific data representations, low-level reasoning processes, and possible tool combinations. It is worth mentioning that the brokering mechanism is domain-independent in the sense that the only domain-specific knowledge is encoded in our ontology. There is no knowledge about proof systems hardwired in our broker. Furthermore, our approach is not limited to a first-order logic domain. With an appropriate extension of our ontology we hope to be able to describe also higher-order theorem provers.

Several frameworks for the integration of reasoning systems have been developed. The MathWeb Software Bus [27] (MathWeb-SB) integrates heterogeneous reasoning systems at the system level. However, the user of the MathWeb-SB needed to have quite a lot of knowledge of the systems involved. Furthermore, a dynamic combination of systems by the MathWeb-SB is not possible. The question of how theorem proving components can be easily combined in a single environment has led to the concept of Open Mechanized Reasoning Systems [11] (OMRS). In OMRS, systems are described on three different levels: the control, the logic, and the interface level. These descriptions are far more fine-grained than the service descriptions used in this work, because they aim at a low-level corporation of systems. OMRS has been studied mainly for the combination of theorem provers and decision procedures.

Future work includes a refinement of the existing services, the incorporation of new services and a more sophisticated brokering mechanism. At this stage Otterfier, for instance, is capable of transforming a derivation only if all the inference steps produce a logical consequence. As indicated in Section 2.2, many modern ATP systems use some form of satisfiability preserving splitting rule, which is useful in the context of a search for a refutation. Otterfier cannot transform derivations containing applications of splitting. In the future it is planned to build a transformation tool that will remove splitting steps from a derivation, by “glueing” together the parts of the derivation that contain clauses inferred by splitting.

So far, our focus has been on first order theorem provers and proof transformation services. In the future it would be desirable to integrate higher-order proving systems, model generators, and decision procedures into the framework. The formalization of systems' logics and calculi in the Logical Framework (LF), implemented in the Twelf [17] system, is being considered.

The brokering of services can be improved in several ways. Reasoning on our ontology during plan formation, for instance, could improve the flexibility of our broker. The use of several plans, conditional plans, and re-planning, could improve the broker's behavior in case plan execution fails.

Acknowledgments

Thanks to Christoph Benzmüller, Serge Autexier, and Armin Fiedler for their contributions to this work.

Bibliography

[1] P.B. Andrews. Transforming matings into natural deduction proofs. In *Proc. of CADE-5*, pages 281–292, 1980.

[2] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[3] O. Caprotti and W. Schreiner. Towards a mathematical services description language. In *Proc. of the International Congress of Mathematical Software, ICMS 2002*, Beijing, China, August 2002.

[4] The MKM Consortium. Mathematical Knowledge Management Network. http://monet.nag.co.uk/mkm.

[5] The MONET Consortium. The MONET Project. http://monet.nag.co.uk/cocoon/monet/index.html, April 2002.

[6] L. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and Melham T. The PROSPER Toolkit. *International Journal on Software Tools for Technology Transfer*, 4(2):189–210, 2000.

[7] L. Dirat. Jome: The java openmath editor. `http://mainline.essi.fr/wiki/bin/view/Jome/WebHome`.

[8] S. Bechhofer et al. OWL – Web Ontology Language Reference, February 2004. Available at http://www.w3.org/TR/owl-ref/.

[9] A. Fiedler. Using a cognitive architecture to plan dialogs for the adaptive explanation of proofs. In Thomas Dean, editor, *Proc. of the 16th International Joint*

Conference on Artificial Intelligence (IJCAI), pages 358–363, Stockholm, Sweden, 1999. Morgan Kaufmann.

[10] A. Fiedler. *P.rex*: An interactive proof explainer. In Rejeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning — 1st International Joint Conference, IJCAR 2001*, number 2083 in LNAI, pages 416–420, Siena, Italy, 2001. Springer Verlag.

[11] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning theories – towards an architecture for open mechanized reasoning systems. IRST-Technical Report 9409-15, IRST, Trento, Italy, Juni 1994.

[12] The Calculemus Interest Group. The CALCULEMUS Project. `http://www.eurice.de/calculemus/`.

[13] X. Huang. Reconstructing proofs at the assertion level. In *Proc. of CADE-12*, pages 738–752, 1994.

[14] SRI International Computer Science Laboratory. QED Pro Quo. `http://www.qpq.org`.

[15] W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[16] A. Meier. TRAMP: Transformation of Machine-Found Proofs into Natural Deduction Proofs at the Assertion Level. In *Proc. of CADE–17*, volume 1831 of *LNAI*, pages 460–464. Springer, 2000.

[17] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proc. of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[18] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[19] P. Rudnicki. An Overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–332, 1992.

[20] W. Schreiner and O. Caprotti. The MathBroker Project. http://poseidon.risc.uni-linz.ac.at:8080/index.html, October 2001.

[21] S. Schulz. System Abstract: E 0.61. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proc. of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 370–375. Springer-Verlag, 2001.

[22] G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In Alan Bundy, editor, *12th International Conference on Automated Deduction, CADE-12*, volume 814 of *LNAI*, pages 252–266, Nancy, France, Juni 1994. Springer Verlag, Berlin.

[23] G. Sutcliffe, J. Zimmer, and S. Schulz. Communication Standards for Automated Theorem Proving Tools. In V. Sorge, S. Colton, M. Fisher, and J. Gow, editors, *Proc. of the Workshop on Agents and Automated Reasoning, 18th International Joint Conference on Artificial Intelligence*, 2003.

[24] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Tpoic. System Description: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proc. of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 378–382. Springer-Verlag, 1999.

[25] L. Wos and G. Pieper. *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. Rinton Press, 2003.

[26] J. Zimmer. A Framework for Agent-based Brokering of Reasoning Services. In Raul Monroy, Gustavo Arroyo-Figueroa, Luis Enrique Sucar, and Juan Humberto Sossa Azuela, editors, *MICAI*, volume 2972 of *Lecture Notes in Computer Science*. Springer, 2004.

[27] J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In A. Voronkov, editor, *Proc. of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 139–143. Springer-Verlag, 2002.

English Summaries of Mathematical Proofs

Marianthi Alexoudi[1], Claus Zinn, Alan Bundy
Division of Informatics, The University of Edinburgh

Abstract

Automated theorem proving is becoming more important as the volume of applications in industrial and practical research areas increases. Due to the formalism of theorem provers and the massive amount of information included in machine-oriented proofs, formal proofs are difficult to understand without specific training. A verbalisation system, ClamNL, was developed to generate English text from formal representations of inductive proofs, as produced by the Clam proof planner. The aim was to generate natural language proofs that resemble the presentation of proofs found in mathematical textbooks and that contain only the mathematically interesting parts of the proof.

1 Introduction

Automated theorem proving is becoming more important as the volume of its applications increases. It is a powerful tool for hardware design, as well as for the verification of software systems. Additionally, formal methods are increasingly applied in mathematics e.g. for the construction of proofs for conjectures and the composition of formalised theories.

Machine-generated proofs are difficult to understand without specific training and familiarity with the given system's formalism and the calculus being used. They are represented in a specialised and artificial language, using a notation that seems incomprehensible to an inexperienced reader. Furthermore, the massive amount of information presented in a formal proof leads to an over-detailed, and therefore hardly readable proof, even for an experienced reader. One could also claim that a formal proof is mathematically 'unstructured', in the sense that it is hard to understand its overall logical structure and identify the important definitions, lemmas and other logical dependencies. Therefore machine-found proofs seem to be insufficient for an effective communication between theorem provers and their users, particularly in terms of their presentation.

Due to the 'unreadability' of formal proofs, the task of their verbalisation and the development of systems, whose output could be conveniently comprehended by non-experts and effortlessly explained by experts became evident. In fact, one of the most challenging tasks in the area of automated theorem proving is the realisation of an effective translation of machine-found (formal) into human-oriented (informal) mathematical proofs, and vice versa. Such a translation would eliminate the gap between the mathematicians' and the proof systems' language and reasoning.

[1]M.Alexoudi@sms.ed.ac.uk,{zinn,bundy}@inf.ed.ac.uk

This paper presents an implemented proof presentation system that generates Natural Language (NL) proofs at various levels of abstraction from (inductive) proof plans. In section 2, related systems are introduced and their weaknesses are briefly discussed. Section 3 provides an overview of ClamNL and examples of its output are presented. A summary of the experimental results is presented in section 4 and the current state and further work on this project are discussed in section 5. Finally, section 6 concludes.

2 Literature Survey

Several efforts have been made to improve the readability of machine-oriented proofs by generating (English) NL versions of proofs. Numerous systems have been designed and developed to produce informal proofs from formal ones that were produced using various deduction techniques and calculi, such as Natural Deduction (ND), resolution and λ-calculus.

Previous work can be classified into three main categories with respect to the output that the existing systems have generated. The first category involves the first generation of verbalisation systems, such as EXPOUND [5] and χ-proof [9] that generated low-level NL proofs, definitely more readable and coherent from machine-found ones but still obscure. Additionally, both Coq [6] and ILF [7] theorem proving systems have a NL front-end that allows the generation of (pseudo)-NL proofs. Although these systems use different methods to generate natural language proofs, most of them suffer from the same problem. The NL versions of machine-oriented proofs were produced by translating a great number of low-level steps, and thus they contained 'obvious' and unnecessary information, such that even trivial proofs might be confusing. Furthermore, most of the informal proofs produced by the above systems preserved the logical formulae in their original form and text was inserted between them either in the form of introductory phrases or as explanations of the inference rules. Thus, the NL proofs are characterised by a mixture of formal and informal representation of the proof steps that reduces their readability. The second group comprises systems such as PROVERB [12] and the NLG module of the Nuprl theorem prover [11], that used more composite and sophisticated techniques to eliminate the drawbacks of previous ones and generated more abstract and human-like NL proofs. Regardless of the sophisticated methods used by these systems, their output is still restricted in some aspects. These systems produce a unique NL version of the corresponding machine-generated proof at a fixed level of abstraction independent of the reader's knowledge. Their output might be too advanced for novice users and too elementary for experts, since it assumes a certain audience with specific knowledge and it does not allow shifting between multiple abstraction levels. The last category embodies systems such as THEOREMA [2] and P.rex [10] that are capable to output various informal proofs for a single formal one.

An essential feature of proofs that enhance their readability is the resemblance to human-written proofs and especially to those written by mathematicians, in terms both of content and presentation. However, the attempts made mostly focus in resembling the way that mathematicians write their proofs, rather than the way that mathematicians reason during proof construction. In many cases this is an issue that arises from the prover rather than the proof presentation system. The formal language and the deduction techniques used for the construction of a formal proof, not only limit the degree of similarity between informal and textbook proofs, but also restrict the level of abstraction and the readability of an NL proof. More precisely, resolution calculi based formal proofs are difficult to manipulate in order to produce coherent NL proofs, due to the existence of a single calculus rule. As far as the ND calculus is concerned, although ND proofs have more potential than resolution ones, it is still complex to abstract the important proof steps from low level inference rules. On the other hand, it is more likely to generate comprehensive and easy readable NL versions of formal proofs produced by tactic-based environments, since related inference rules are grouped into tactics, each of which approximates a single human inference step.

Therefore, aiming at the construction of informal proofs similar to those presented in mathematical textbooks, we need to use formal proofs resembling the way that mathematicians analyse and work out proofs. For instance, mathematicians recognise families of proofs containing common structure and they use previously encountered proofs to assist them in discovering new ones. The way that mathematicians work out their proofs can be captured using the proof planning technique for constructing and representing high-level proofs [3]. Proof plans are abstract representations of proofs at a level that is better suited for manipulation beacuse of the absence of low-level derivations.

3 System Overview

ClamNL [1] is a proof presentation system built upon the Clam proof planning environment [4] that generates NL proof at various levels of abstraction, similar to those found in mathematical textbooks. Clam is a first-order predicate logic proof planner that was used for the construction of proof plans for theorems whose proofs require the application of various kind of mathematical induction over different data types.

The use of high-level representations of proofs, known as proof plans, enhances the generation of abstract NL proofs. Furthermore, the process of formal proof conversion is informed by a notion of 'interestingness' of proof steps. A set of heuristics has been employed to remove obvious and trivial parts of the proof and highlight its mathematically interesting points.

ClamNL's architecture is presented in Figure 1. Of the modules illustrated in Figure 1, the proof planner (Clam) and the XSLT-based software (Natural Language Generator) that processes the templates are existing software. Each of the remaining components is described in one of the following sections. ClamNL consists of three main modules, the *Abstraction Controller* that enables the interaction of the system with the user and the proof planner; the *Structure Planner* that handles the structure, the contents and the presentation of the NL proofs to be generated; and the *NL Generator* that translates the extracted parts of a proof plan to English text in a template-based manner.

3.1 Abstraction Controller

The Abstraction Controller (AC) controls the level of detail of a proof plan and in turns the level of abstraction of the resulting NL proof. The AC, given a theorem name, determines the number of proof versions that can be generated for that theorem.

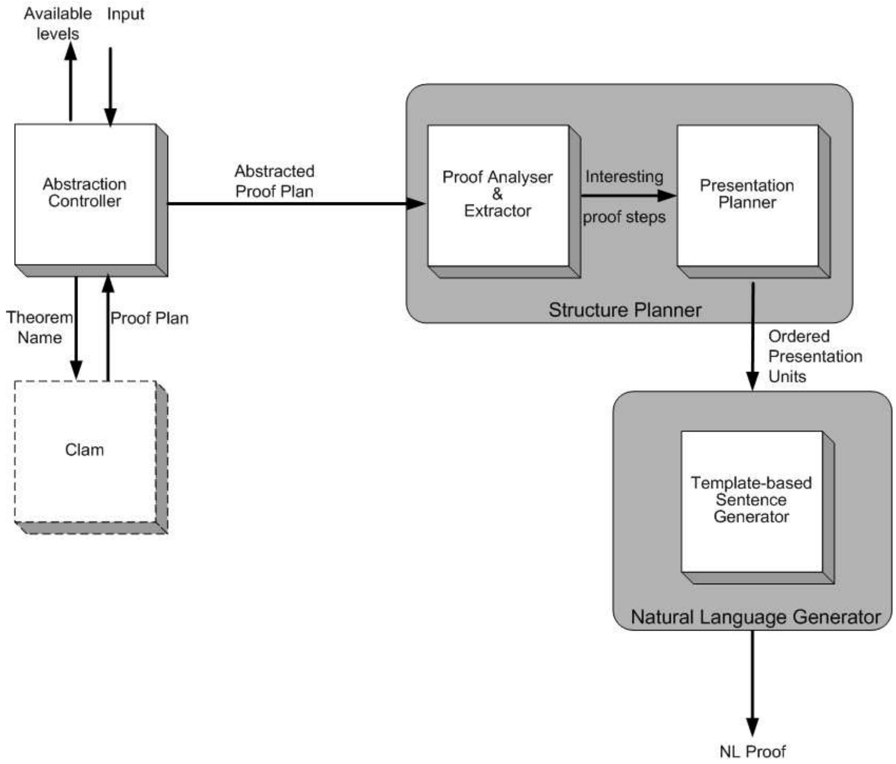


Figure 3.1: Architecture of the ClamNL verbalisation system

Moreover, given a theorem name and the required level of abstraction, it initialises the process of generating the requested NL proof.

The process of abstracting a proof plan depends entirely on the theorem to be proven. If the proof of a theorem involves the proof of another theorem, then the AC discards from the original proof plan the subproof and passes the remaining proof plan to the Proof Analyser and Extractor (PAE). This process can be repeated as many times as the total number of theorems used to prove the original one. Therefore, the number of NL proofs at different levels of abstraction depends on the number of theorems that are used in the proof of the original one. In this case the resultant NL proof presents the proof of the original theorem, in which other theorems are used for its completion. Although none of them is proven, it is assumed that they hold.

3.2 Proof Analyser and Extractor

The Proof Analyser & Extractor PAE determines the nodes of an abstracted proof plan to be included into a NL proof. Given an abstracted proof plan, PAE extracts the mathematically interesting parts of a proof and omits standard and easily deducible ones. During the extraction of interesting proof steps the proof plan tree is linearised and every node (subproof) is handled separately. The output of PAE is a forest of proof steps that is then passed to the Presentation Planner. An example of such transformation is presented in Figure 2.

Input: Abstracted proof plan

```
problem([]==>x:pnat=>y:pnat=>z:pnat=>plus(x,plus(y,z))=plus(y,plus(x,z))in pnat,
    quantifier_to_front(x:pnat)
    then try dequantify_once
    then do_induction([(x:pnat)-s(v0)]),
    [problem([x:pnat]
        ==>y:pnat=>z:pnat=>plus(0,plus(y,z))=plus(y,plus(0,z))in pnat,
        normalize_term([reduction([1,1],[plus1,equ(pnat,left)]),
                reduction([2,2,1],[plus1,equ(pnat,left)])])
       then[elementary((intro(new[y])then wfftacs)
                then(intro(new[z])then wfftacs)
                then unfold_iff then identity)],[]),
    problem([x:pnat,v0:pnat,v1:y:pnat=>z:pnat=>plus(v0,plus(y,z))=plus(y,plus(v0,z))in pnat]
        ==>y:pnat=>z:pnat=>plus(s(v0),plus(y,z))=plus(y,plus(s(v0),z))in pnat,
        ripple(direction_out,wave(direction_out,[1,1],[plus2,equ(pnat,left)],[])
                 then[wave(direction_out,[2,2,1],[plus2,equ(pnat,left)],[])])
        then[unblock_then_fertilize(weak,unblock_fertilize_lazy([idtac])
                    then fertilize(weak,pwf_then_fertilize(weak,fertilize_left_or_right([
                                weak_fertilize(left,in,[1],v1)]))then[idtac]))],
```

Output: Extracted proof steps

```
problem(x,y,z,plus(x,plus(y,z))=plus(y,plus(x,z)),pnat)
do_induction(primitive_ind,x)
subproof(assumption(x),exp(plus(0,plus(y,z))=plus(y,plus(0,z))))
normalize_term(plus1,plus1)
subproof(assumption(v0,plus(v0,plus(y,z))=plus(y,plus(v0,z))),
exp(plus(s(v0),plus(y,z))=plus(y,plus(s(v0),z))))
ripple(plus2,plus2)
fertilize
```

Figure 3.2: Extraction of mathematically interesting proof steps from a proof plan.

The 'interestingness' of proofs steps is defined by a knowledge base according to which a proof step can be interesting, non-interesting, or partially interesting.

In general, apart from the theorem statement, the focus of interest in inductive proofs is on the induction scheme used to prove the theorem and the induction variable to which the induction scheme is applied. Also, the base and the step cases of an inductive proof should be clearly stated. In the base case, the induction variable and the constant to which the induction variable is instantiated, as well as the base case resulting expression should be specified. Similarly, in a step case, the induction hypothesis and the conclusion should also be specified. On the other hand, axioms and low-level methods are classified as mathematically non-interesting proof steps and thus are discarded. Moreover, rewriting on conjectures that produce the same conjecture as the one to which they were applied should be ignored, as well as their resultant conjecture (i.e. tautology). As partially interesting are charecterised proof steps whose some of their contents are useful and some of them are not. An example of this category is the list of hypotheses available every time a new goal is introduced, which might contain a new assumption.

3.3 Presentation Planner

The Presentation Planner (PP)performs three vital tasks. It rearranges the contents of the proof steps, inserts additional elements where appropriate, and transforms the proof steps to an intermediate representation format consisting of presentation units. The ordered presentation units are then passed to Natural Language Generator module to be verbalised.

The PP features two stages of reordering the contents of proof steps but not the proof steps themselves. The first involves the checking of the proof steps ordering, in case their order was lost during linearisation, and the reordering of the terms of a mathematical formula from infix to prefix. The second involves the mapping of certain compound terms of a conjecture into a more human-oriented representation and the ordering of the new terms in the conjecture.

One feature involves the random selection of justification tokens that will be used for the verbalisation of certain units. The Presentation Planner handles, in a non-sophisticated way, commonly used tokens to avoid the repetition of identical standard phrases in the proof outline. Every justification token corresponds to a single sentence in the template-based sentence generator. In principal, this involves the identification of base and step cases proof steps in order to avoid incorrect verbalisations. Another approach to avoid multiple interesting proof steps in the proof outline involves the merging of similar, adjacent proof steps.

Finally, the presentation of a proof in enhanced by organising the proof steps into paragraphs and indenting them so that the proof structure is clear and thus coherent. Also, an axiom table, consisting of all the axioms' definitions used throughout the proof, is appended to the end of the proof. During the structuring of the proof steps, Presentation Planner (PP) converts the proof steps to XML elements and produces the XML document to be fetched to the NLG module.

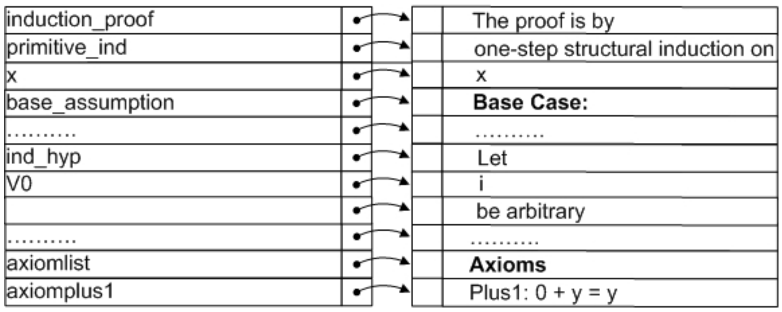


Figure 3.3: An example of presentation units to NL templates mapping.

3.4 NLG

The template-based generator maps presentation units to English statements. Text is generated by mapping individual presentation units (XML elements) into mathematical notions, concepts, variable names, words and sentences (XML templates). Figure 3 illustrates a sample of such a mapping. The Template-based generator, given an input XML document, outputs a NL proof in a HTML or XML file format.

As a tool for the text generation, a major component of the EXEMPLARS framework [14], the *text builder* was used as a wrapper for the XSLT text building transformation.

Also, a XSL stylesheet consisting of more than 400 templates was created for the mapping of the XML elements to English text. The XSLT engine matches the template rules contained in the XSL stylesheet with the XML elements of the input document and the text is generated, based on the XSL vocabulary.

NL proofs such as the one presented in Figure 4 consist of the theorem statement and the outline of the proof. The actual proofs are constructed using a Lamport-style proof presentation [13], consisting of the theorem statement, the proof outline and the proof of the theorem.

3.5 Sample Output

The commutativity of addition is provided, as an example of the system's output for illustration purposes. The proof of the provided theorem is an example of a proof by induction over natural numbers, one of the various induction schema available in Clam. Others include one and two step induction on lists and trees.

Figure 4 presents an example of a NL version of a machine-found proof corresponding to the commutativity of addition theorem, generated by ClamNL. This kind of output is called a proof summary, since it states how a certain theorem can be proven. More precisely, it is more like a proof description about what one should do in order to prove a theorem, rather than a proof itself.

Theorem: (x + (y + z)) = (y + (x + z)) for all natural numbers x y z
The proof is by one-step induction on x In order to complete the step case, the conjecture is generalised by introducing a universal variable i.e. k . The generalised conjecture can be proven by one-step induction on y and finally the proof is completed

Figure 3.4: The proof summary of the commutativity of addition theorem.

The actual proof of the commutativity of addition theorem, but more abstract than the complete one is presented in Figure 5. Such kinds of proofs involve the progress of the proof until the stage where another known theorem is used to complete the proof of the actual theorem.

Figure 6 [2] shows a more detailed proof version of the commutativity of addition theorem. Although the complete proof of the theorem is presented, it is not a direct translation of the original, machine-found proof plan, since lots of proof steps and trivial parts of the proof are omitted.

4 Experimental Results

ClamNL was developed to generate NL versions of formal mathematical proofs that would be fluently readable and abstract and would resemble those found in mathematical textbooks. To demonstrate and validate the project's claims, a corpus of inductive

[2] In certain cases the template mapping produces minor grammatical errors, whose elimination is in high priority

Theorem: (x + (y + z)) = (y + (x + z)) for all natural numbers x y z
The proof is by one-step structural induction on x
Base Case: x = 0
(0 + (y + z)) = (y + (0 + z)) by applying 2 times axiom "plus1"
Inductive Hypothesis: Let i be arbitrary
(i + (y + z)) = (y + (i + z))
Induction Step: We need to prove ((i + 1) + (y + z)) = (y + ((i + 1) + z))
by applying 2 times axiom "plus2" and using the induction hypothesis we get:
((y + (i + z)) + 1) = (y + ((i + z) + 1))
Let: k = (i + z)
we are left to prove: ((y + k) + 1) = (y + (k + 1))
Since by plus2right theorem, we have that (x + (y + 1)) = (x + y) +1 is true,
then (x + y) +1 = (x + (y + 1)) is also true. Thus, the proof is completed.

Axioms
plus1: 0 + y = y
plus2: (x + 1) + y = (x + y + 1)

Figure 3.5: A NL proof of the commutativity of addition theorem at an intermediate level of abstraction

theorems was collected from approximately 130 supported theorems. The corpus was selected so as to provide a wide range of theorems covering various degrees of difficulty and complexity, as well as the use of various different induction schema.

Two groups of subjects were used, Clam experts and non-experts with different mathematical background, in order to ensure that the NL proofs were accessible and beneficial to a wide range of audience with various levels of expertise. Although the number of participants was too limited to obtain a statistically representative sample, the results gathered were beneficial and encouraging.

The results are classified into three categories, each of which corresponds to and supports the project's objectives.

Readability: The NL proofs were characterised as easily readable and coherent on two counts. First, in terms of the linguistic nature of the proofs, since it is considerable easier for a human to comprehend a proof in the (natural) language of mathematicians, rather than rules represented in the given proof system's formalism. Second, the use of indentation was quite helpful in keeping track of deeply nested proofs (i.e. subproofs by induction). However, in cases of deeply nested proofs it would be preferable to have a hypertext-based or applet-based approach to hide or unfold parts of the proof presentation on the fly upon user requests.

Abstraction Level: The availability of proofs at various levels of detail was found extremely useful in digesting proofs. As regards the content of the NL proofs at different levels of abstraction, different opinions were expressed, possibly because of the dissimilar mathematical background of the participants. The majority of the subjects claimed that each proof contained the right amount of information on the progress of the proof, given the corresponding detail level. In particular, proofs declared as abstract were indeed seen

Theorem: $(x + (y + z)) = (y + (x + z))$ for all natural numbers x y z
The proof is by one-step structural induction on x
Base Case: $x = 0$
$(0 + (y + z)) = (y + (0 + z))$ by applying 2 times axiom "plus1"
Inductive Hypothesis: Let i be arbitrary
$(i + (y + z)) = (y + (i + z))$
Induction Step: We need to prove $((i + 1) + (y + z)) = (y + ((i + 1) + z))$ by applying 2 times axiom "plus2" and using the induction hypothesis we get:
$((y + (i + z)) + 1) = (y + ((i + z) + 1))$
Let: $k = (i + z)$ we are left to prove:
$((y + k) + 1) = (y + (k + 1))$
The proof is by one-step structural induction on y

> **Base Case:** $y = 0$
> $((0 + k) + 1) = (0 + (k + 1))$ by applying 2 times axiom "plus1"
> **Inductive Hypothesis:** Let o be arbitrary
> $((o + k) + 1) = (o + (k + 1))$
> **Induction Step:** We need to prove $(((o + 1) + k) + 1) = ((o + 1) + (k + 1))$ by applying 2 times axiom "plus2" and using the induction hypothesis the case is completed.

Axioms
plus1: $0 + y = y$
plus2: $(x + 1) + y = (x + y + 1)$

Figure 3.6: The most detailed proof version of the commutativity of addition theorem.

as abstract and those declared as detailed were indeed seen as more detailed. However, there were cases where participants claimed that the omission of some steps would be desirable, though their presence was not irritating.
Similarity to textbook proofs: The participants, based on their experience in mathematical literature, estimated that the presentation of the NL proofs approximated the presentation of proofs found in mathematical textbooks.

5 Current State and Further Work

ClamNL has been developed as the first author's undergraduate project. It offers many opportunities for improvements and extensions, which we will discuss next.

The next step in the development of NL versions of machine-oriented proofs would be the generation of partial NL proofs of unsuccessfully proven theorems. Proof presentation systems developed so far, require their respective theorem provers to compute complete proofs. Our idea is to verbalise a formal proof until the point that it has been successfully developed and then try to explain in natural language the reasons that lead to a fallible proof, and if possible, suggest patches about how a certain failure can be overcome in natural language.

Currently, we are also investigating to adapt our verbalisation system so that it can handle IsaPlanner[8]-generated proof plans. IsaPlanner is a generic framework for proof planning build upon the interactive theorem prover Isabelle that facilitates reasoning techniques to conjecture and prove theorems automatically. In constrast to Clam that limits the implementation of partial NL proofs, IsaPlanner supports the generation of both complete and incomplete proof plans. Furthermore, IsaPlanner is available to a wider audience compared with Clam, which is nowdays used by a limites number of people.

As far as the verbalisation of complete proofs is concerned, the current system generates various abstract NL proofs that contain only mathematically 'interesting' parts of the proof, rather than the complete proof in terms of low-level steps. At the moment, the 'interestingness' of proof steps is system-defined. Thus, the next step would be to modify it so that users would be able to obtain customised versions of NL proofs by defining what they consider to be interesting, depending on their knowledge and interest.

The provision of user interaction is an important feature in systems of this nature. The user can interact with the system either by requesting the number of available NL versions of the proof of some theorem or by requesting a certain proof of a theorem. However, the interaction between the user and the system is managed through a unix shell. This is clearly a limitation that we would like to resolve in future by designing a simple and user-friendly interface. Furthermore, although the proofs are expandable, in the sense that some are more detailed than others, hypertext-based versions of proofs would be extremely useful, since users could unfold parts on the fly rather than look for another version of the proof.

Finally, additional features would be the gen eration of multilingual proofs and proofs of different presentation styles targeted at two different groups of users. A *mathematical-style* for users that are interested in mathematical aspects of the proof of a given theorem and a *compositional-style* that will target people interested in the process of constructing proofs.

6 Conclusion

This paper presents and proposes a multi-step approach for the presentation of machine-oriented proofs. The automatic generation of NL versions of formal proofs aims to improve the readability and comprehensiveness, as well as the usefulness of machine-found proofs and extend/enable their availability to a wider audience.

The generation of human-readable proofs at different levels of abstraction can be succeeded using the proof planning technique. Proof plans offer an ideal solution, since they provide high-level presentation and low-level interpretation of proofs. In addition, the process of a formal proof abstraction and the availability of proofs at different levels of details is enhanced by the notion of interestingness. Currently, the interestingness of proof steps is system defined, but in future we aim to a more dynamic and interactive approach to proof presentation and explanation.

Bibliography

[1] M. Alexoudi. English Summaries of Mathematical Proofs, 4th Year Undergraduate Project Report. School of Informatics, University of Edinburgh, 2003.

[2] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuţa, and D. Văsaru. A Survey on the THEOREMA Project. In *ISSAC '97. Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 384–391. ACM Press, 1997.

[3] A. Bundy. A Science of Reasoning. In *J. L. Lassez & G. Plotkin (Eds.), Computational Logic: Essays in Honor of Alan Robinson*. The MIT Press, 1991.

[4] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam System. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 647–648, Kaiserslautern, FRG, July 1990. Springer Verlag.

[5] D. Chester. The Translation of Formal Proofs into English. *Artificial Intelligence*, 7:261–278, 1976.

[6] Y. Coscoy. A Natural Language Explanation for Formal Proofs. In *Proceedings of the 1st International Conference on Logical Aspects of Computational Linguistics (LACL-96)*, volume 1328 of *LNAI*, pages 149–167, Berlin, September 23–25 1997. Springer.

[7] B. I. Dahn and A. Wolf. Natural language presentation and combination of automatically generated proofs. In *Frontiers of Combining Systems (FroCos)*, pages 175–192, 1996.

[8] L. Dixon and J. D. Fleuriot. IsaPlanner: A Prototype Proof Planner in Isabelle. In *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2003.

[9] A. Felty and D. Miller. Proof Explanation and Revision. Technical Report MS-CIS-88-17, University of Pennsylvainia, 1987.

[10] A. Fiedler. *P.rex: An Interactive Proof Explainer*. In Rejeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning — 1st International Joint Conference, IJCAR 2001*, number 2083 in LNAI, pages 416–420, Siena, Italy, 2001. Springer Verlag.

[11] A. M. Holland, R. Barzilay, and R. L. Constable. Verbalization of High Level Formal Proofs. *In American Association for Artificial Intelligence*, 1999.

[12] X. Huang and A. Fiedler. Presenting machine-found proofs. In *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volume 1104 of *LNAI*, pages 221–225, Berlin, July 30–August 3 1996. Springer.

[13] L. Lamport. How to Write a Proof. Technical Report TR 94, Digital Systems Research Center, February 1993.

[14] M. White and T. Caldwell. EXEMPLARS: A Practical, Extensible Framework for Dynamic Text Generation. In Eduard Hovy, editor, *Proceedings of the Ninth International Workshop on Natural Language Generation*, pages 266–275. Association for Computational Linguistics, New Brunswick, New Jersey, 1998.

The Use of Data-Mining for the Automatic Formation of Tactics

HAZEL DUNCAN[1] ALAN BUNDY JOHN LEVINE AMOS STORKEY[2]
SCHOOL OF INFORMATICS, UNIVERSITY OF EDINBURGH, UK

MARTIN POLLET[3]
FACHRICHTUNG INFORMATIK, UNIVERSITÄT DES SAARLANDES, GERMANY

Abstract

The aim of this project is to evaluate the applicability of data-mining techniques to the automatic formation of tactics from large corpuses of proofs. We data-mine information from large proof corpuses to find commonly occurring patterns. These patterns are then evolved into tactics using genetic programming techniques.

1 Motivation

Within the field of automated deduction, the huge search spaces involved in finding correct proofs means that fully automated theorem provers are not as advanced as it was once thought they would be by this time. For example, Newell and Simon claimed that a computer would "discover and prove an important new mathematical theorem" by January 1st 1968 [Newell & Simon]. An important advance in theorem proving was made by Robin Milner when he introduced the notion of tactics. The introduction of tactics helped the field of theorem proving by guiding search, this project aims to build upon that success by implementing a method to allow tactics to be formed automatically. Tactics are functions from goals to subgoals whcih can fail raising appropriate error messages. Robin Milner used tactics in his automatic proof assistant theorem prover Edinburgh LCF [Milner], which initiated interactive theorem-proving and the proof-assisting tradition. LCF has led to descents such as HOL, ISABELLE, COQ and LEGO.

In interactive theorem proving systems, a theory is viewed as collection of definitions, theorems and proofs, as well as tactics. An important aspect for the development of a theory is therefore not only the formalisation of a theory, but also the reasoning techniques for proofs, that is, the tactics. With this project we hope to contribute to the development of theories by providing a way to automate the formation of tactics using data-mining, we hope this would lead to new tactics for more complex theorems. This would then, in turn, aid the development of the formalisation of theories.

In order to do this, usable proof corpuses have been chosen and transformed into a suitable format for the data-mining.

[1]Work supported by the European Commission IHP Calculemus Project grant HPRN-CT-2000-00102 and EPSRC grant GR/S76328/01

[2]Work supported by a fellowship from Microsoft Research, Cambridge.

[3]Work supported by the European Commission IHP Calculemus Project grant HPRN-CT-2000-00102

We have adapted probabilistic reasoning techniques, such as Variable Length Markov Models (VLMM) [VLMM] and Log Linear Models (LLM) [LLM], to the identification of rule sequences that are useful for predicting the next rule. These techniques are used to discover patterns existing in the proof corpuses.

These patterns can be viewed as simple tactics, which are adapted using Koza-style genetic programming [Koza]. Using this, we will generalise these simple tactics into more complex ones, e.g. containing repetition and branching. This requires the development of an evaluation function for scoring the evolving tactics.

The new generalised tactics will be evaluated, e.g. by applying them to a set of test theorems and comparing their performance to the available alternatives.

2 Related Work

There have been several previous attempts to learn new proof methods or tactics from example proofs. Also of interest to us are systems which learn and predict patterns, this has been particularly common in the bioinformatics community.

In his PhD project at Edinburgh, Bernard Silver applied techniques of explanation-based learning to the automated learning of proof methods for equation solving [Silver]. His Learning-Press system analysed successful solutions to equations and generalised these solutions to form methods for guiding the Press equation solving system. In this way, he was able to automatically rediscover simplified versions of many of the previously hand-coded methods of Press.

Ron, Singer and Tishby created a distribution learning algorithm for Variable memory Length Markov Models [Ron *et al*]. These processes can be described as a subclass of probabilistic finite automata (PFA). Though hardness results are known for learning distributions generated by general probabilistic automata, they prove that the algorithm they present can efficiently learn distributions generated by PFAs. In particular, they show that, for any target PFA, the KL (Kullback-Liebler)-divergence between the distribution generated by the target and the distribution generated by the hypothesis the learning algorithm outputs, can be made small with high confidence in polynomial time and sample complexity. The learning algorithm is motivated by applications in human-machine interaction. As with our project, they looked at data which has a *short memory property*, i.e., consider the empirical probability distribution on the next symbol in a sequence given the preceding symbols, then there exists a length L (*memory length*) such that the conditional probability distribution does not change substantially if we condition on preceding subsequences of length greater than L. These can form Markov models of order L>1, they give efficient procedures both for generating sequences and for computing their probabilities.

More recently, Kerber, Jamnik, Pollet and Benzmüller have applied the techniques of least general generalisation to a family of similar proofs to learn new proof methods for various domains [Kerber *et al*]. They present a technique for automated learning within mathematical reasoning systems. In particular, this technique enables proof planning systems to automatically learn new proof methods, from well chosen examples of proofs that use a similar reasoning pattern, to prove related theorems. Their technique consists of a representation formalism for methods and a machine learning technique which can learn methods using this representation formalism. They present an implementation of this technique, called LearnΩMatic, which adds new methods to the Ωmega proof planner. Methods are represented using a regular grammar over individual proof steps and previously learned methods, allowing a hierarchical collection of methods. Note that this technique requires all the proofs in the family to be examples of the learned method.

John Levine and David Humphreys [Levine & Humphreys] developed L2Plan (learn to plan), a genetic programming based method for planning. Their system represents control knowledge as a *policy* and learns using Genetic Programming. The program's crossover and mutation operators are augmented by simple local search. L2Plan was able to produce policies which solved all the test problems it was given, outperforming hand-coded policies written by the authors. The genetic programming used for this would be well suited to our task, randomly generating an initial population and then evaluating their fitness against our test set may well produce results that would be difficult to find using other methods.

3 Outline

Automatic learning by reasoning systems is a difficult and ambitious problem. Our work demonstrates one way of starting to address this problem, and by doing so, it makes several contributions to the field.

1. Although machine learning techniques have been around for a while, they have been relatively little used in reasoning systems. Making a reasoning system learn proof patterns from examples, much like students learn to solve problems from examples demonstrated to them by the teacher, is hard. Our work makes an important step in a specialised domain towards a proof planning system that can reason *and* learn.

2. Proof methods have complex structures, and are, hence, very hard to learn by the existing machine learning techniques. We approach this problem by abstracting only as much information from the proof method representation as needed, so that the machine learning techniques can handle the information. Later, after the reasoning pattern is learnt, the abstracted information will be restored to its original form as much as possible.

3. Unlike in some of the existing related work, we are not aiming to improve ways of directing proof search within a fixed set of primitives. Rather we aim to learn the primitives themselves, and to investigate whether this improves the framework and reduces the search space within the proof planning environment. Instead of searching amongst numerous low level proof methods, a proof planner can search with a newly learnt proof method which encapsulates several of these low level primitive methods. Solely using new proof methods is unlikely to be complete, however, if old proof methods are left in place then the search space will in actual fact grow. This suggests that using heuristics to allow new methods to be used first may help find a proof in a shorter space of time despite a larger search space.

4. Our work is also more general than the LearnΩMatic approach, because it does not require a careful manual selection of candidates.

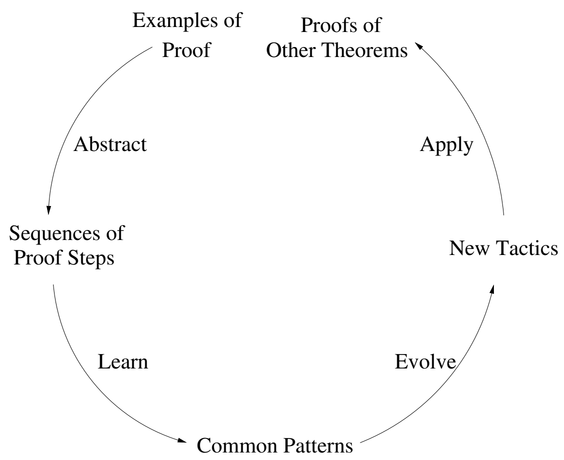


Figure 3.1: The structure of our approach to learning proof methods.

Figure 3.1 represents the structure of our approach. We begin with a large proof corpus of existing proofs which we abstract to sequences of proof steps. We learn commonly occurring patterns from these abstracted proofs using data-mining and probabilistic techniques. These patterns are evolved using genetic programming to form new tactics which can be applied to find proofs of other theorems.

4 Data-Mining the Proof Corpuses

4.1 Choosing the Corpus

A suitable corpus of proofs to be used in this project has been chosen to meet precise requirements. The option remains to include other corpus(es) at a later date.

1. It is stored in electronic form, so that it is available for Data-Mining

2. It is sufficiently large to contain many examples of multiply occurring patterns of proof

3. There is an appropriate diversity of kinds of proof steps, i.e., sufficiently different kinds of proof steps that patterns can be identified, but not so much diversity that patterns do not recur. Note that the appropriateness of diversity is relative to corpus size: the larger the diversity, the larger the corpus required for the re-occurrence of patterns.

4. The corpus lends itself to a suitable abstraction. We are currently only using the rule name at each proof step. The option remains to include more information.

Note first that proofs generated by resolution-style theorem provers are, unfortunately, mostly unsuitable because of requirement 3 above: typically only one or two rules of inference are used. We could try to differentiate rule applications by the formulae they manipulate, but these formulae are generated during the proof search and are too diverse, e.g. millions of derived clauses. In addition, it has been suggested that interactive theorem provers may be more likely to yield interesting patterns due to the structure that people insert in their proofs. Conversely, it has also been suggested that a wholly automatic theorem prover may yield patterns as it searches for proofs in a deterministic way.

Isabelle is an interactive theorem prover developed at Cambridge [Isabelle] which satisfies the necessary criteria. It also has some inbuilt commands which allow the proof of a theorem to be extracted.

Mizar was developed at the University of Bialystock in Poland as an aid to the development of mathematical articles for formalized maths [Mizar]. Mizar also meets the necessary requirements and has been investigated as a usable corpus.

4.2 Markov Models

Once the proof corpuses have been put into a suitable format, probabilistic reasoning techniques are applied to identify patterns within the proof structures, these patterns form the basis for the new tactics.

Markov Models are a probabilistic technique which calculate the probability of something, (in our case, a proof step), given what appears before. The main advantage of using Variable Length Markov Models (VLMMs) is that the number of things that come before is not fixed, i.e., it has memory of variable length. They seem to offer a good solution to the task of identifying patterns. Using the proof corpus, a VLMM is trained and then used to predict the next proof step. One problem of more basic counting techniques is that longer strings would be prejudiced against due to the fact that they are less likely to appear simply by chance. For example, a 'pattern' of rule A followed by rule B would possibly occur (say) 15 times simply by chance, but a pattern of ABCDCBC should be considered to be significant if it happened to occur 4 times. The very nature of VLMMs means that this problem would be simply dealt with.

Although there is existing software which deals with pattern formation and Variable Length Markov Models the software was very specifically for DNA pattern construction and some experiments and investigation proved that adapting this software would be prohibitively complex.

4.3 Extraction of Patterns

One of the major problems encountered with the pattern discovery so far is the case splits in the proofs. Although many pieces of software exist for identifying patterns in strings (most commonly for DNA sequences, but also for more general strings) which could be adapted for use with proof structures, we have been unable to identify any existing software or unimplemented theoreticized techniques which identify patterns within tree structures. It was suggested that case splits could be ignored or simply treated as a special case, i.e., a "split token", however, the high frequency of occurrence of some sort of branching structure within a proof means that in this case we may well lose many interesting patterns.

The technique decided upon was to split the proofs into separate strings and give weights accordingly, i.e., all the steps at the end of any branch have weight 1, before each split the weights are given as $1/branches * weight\ after\ split$ – so a tree which has 3 two-way splits would have a weight of 1 at the end of each branch, 0.5 on every branch between the last two splits, 0.25 after the first split and 0.125 before there is ever a split point. The treatment of branching is exemplified in Figure 4.2. The result is a list of tuples of the form

$$[[[0.5, A], [0.5, B], [0.5, C], [1, D], [1, F]], [[0.5, A], [0.5, B], [0.5, C], [1, E], [1, G]]] \quad (1)$$

These weights are incorporated simply at the point where the Markov Model is updated. It would be much more elegant to have software which learned Markov models directly from the tree structures but this has not yet been found.

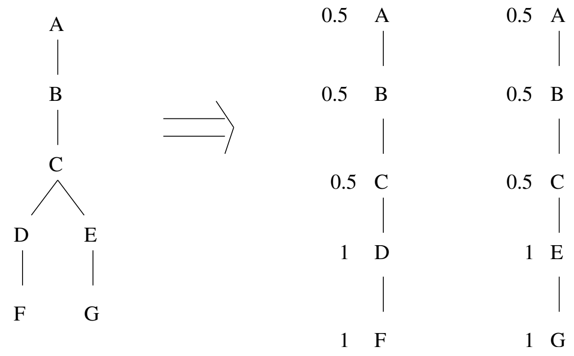


Figure 4.2: Example proof steps with split

The technique currently being used works in a probabilistic-style way, based on the same principles as VLMM. For each point in the proof a number representing the likelihood of that step occurring given the previous steps is calculated. This is calculated by multiplying the number of times that step occurs in the corpus by the weight given to the particular sequence. This means that the sequence $[A, B, C, D]$ is calculated in terms of D. $P(D|[A, B, C]) = O(D) * W$ (where $O(D)$ is the number of times D occurs and W is the weight attributed to D). In such a string $P(D|A)$ and $P(D|[A, B])$ would also be calculated. If any of these strings have occurred before, (for example if $P(D|[A, B, C])$ is already in the database), then the calculated probability is added to the existing probability.

The nature of this software is such that a lot of redundancy is created - however, these redundancies ensure that any sub-part of a discovered pattern which also appears elsewhere will appear as a pattern in its own right as it will have a higher probability.

Finally a threshold is specified and any patterns which have a probability above this threshold are returned. Although any patterns which are directly subsumed by others with the same probability (i.e., they do not appear other than in the longer pattern) are deleted.

These patterns are already sequences which commonly occur within a proof. As such, they already describe a part of the proof and can be thought of as simple probabilistic tactics.

5 Generation of Tactics

5.1 Grammar

Some care is required over the choice of the tactic language. The choice ranges from regular grammars, via a limited set of tacticals to a general programming language, such as ML. A parsimonious language will be better suited to genetic programming, e.g. a limited set of tacticals. Moreover, the language must not require information that cannot be obtained by analysis of the proof corpus. For instance, it is no use including while-loops or if-then-else, if their conditions cannot be identified.

Non-conditional forms of repetition and non-determinism must be used instead. We have therefore decided to represent generalised patterns in the following language $\mathcal{L}$ which is defined as:

$$\begin{array}{ll} t \in \mathcal{L} & \text{for tactic identifiers } t \\ m \in \mathcal{L} & \text{for macro identifiers } m \\ \left.\begin{array}{l} [L_1, L_2] \in \mathcal{L} \\ (L_1 \vee L_2) \in \mathcal{L} \\ (L_1 \wedge L_2) \in \mathcal{L} \end{array}\right\} & \text{for } L_1, L_2 \in \mathcal{L} \\ L\star \in \mathcal{L} & \text{for } L \in \mathcal{L} \end{array}$$

The tactic identifiers denote the tactics which appear in the extracted proof sequences. Macro identifiers are used as abbreviation for a pattern $L \in \mathcal{L}$. The operators have the semantics of tacticals. The term $[L_1, L_2]$ is interpreted as sequencing (L_2 is applied after L_1), $L_1 \vee L_2$ stands for a disjunction (either L_1 or L_2 is applied), and $L_1 \wedge L_2$ has the semantics that L_1 is applied to one subgoal and L_2 to the other subgoal. The term $L\star$ denotes an arbitrary number (greater than one) of repetitions of L.

5.2 Evolving Tactics

The patterns constructed by the probabilistic methods described previously will only consist of fixed length combinations of particular rules. This does not reflect the full generality of hand-built tactics. For instance, recursion might be used to capture the repeated application of a particular sub-tactic a variable number of times. Or conditionals might be used to capture variations in the particular rule combinations. The use of macros allow sub-routines which may occur to be identified.

Generalisation of the extracted sequences is a computationally time consuming problem. In our case it is even more difficult since the result is not expected to be one generalisation describing all the patterns but an unknown number of generalisations which describe different classes of typical patterns. There are two competing criterions to be fulfilled:

1. A generalisation should subsume many sequences.

2. A generalisation should subsume only sequences of one class.

We approached this problem from a different perspective. Instead of trying to find 'the best' set of generalisations which is generated by an computationally expensive algorithm, we gradually produce 'better' generalisations using techniques from genetic programming. This approach implies that the resulting generalisations may be different in different runs and thus could produce generalisations which either subsume many sequences (criterion 1), differentiate between more classes (criterion 2), or lie in between.

To generalise our initial tactics in this way, we use two approaches, a pairwise crossover of our patterns and traditional genetic programming.

Our pairwise crossover of patterns works by randomly choosing two patterns from our discovered patterns and looking for points of combination - suitable points for $\wedge$ branch introduction, candidates for the $\star$ operator, $\vee$ differences in patterns and (where one pattern is completely contained in the other) introduction of macros. New patterns formed are scored against the original pattern set, they are kept if they outperform their predecessors and discarded otherwise. The score for a new pattern is generated with a positive for every old pattern it describes and a negative for members of an $\vee$. This prevents a simple disjunction over all possibilities being accepted as a 'good' tactic.

John Koza explains the principals of Genetic Programming in his book [Koza]. Koza's work describes and illustrates genetic programming with 81 examples from various fields, of particular interest is the 'Evolution of Subsumption', which will be similar to the strategy we wish to use.

This approach genetically breeds populations of computer programs to solve problems by executing three steps:

1. Generate an initial population of random tactics made up of our grammar and our proof step names. Weights are given to the grammar operations and to the likelihood of moving straight on to another step in order to generate a more 'sensible' tactics (a tactic containing only (say) 4 proof steps but 10 of our grammatical operations is unlikely to be useful).

2. Iteratively perform the following sub-steps until the termination criterion has been reached:

   (a) Execute each program in the population and assign it a fitness value

   (b) Create a new population by:

      (i) Reproduction: Copy existing tactics to the new population

      (ii) Crossover: Create two new tactics by genetically recombining randomly chosen parts of two existing tactics

3. The best few tactics at the time of termination is deemed to be the result of the genetic programming. This solution is produced after a time limit. The 'best few' is designated as the fewest high-scoring tactics which completely describe the patterns generated by data-mining.

Although Koza describes his technique in terms of programs, functions and terminals, our technique simply an application of this applied to tactics, proof steps and operations from our grammar. Our scoring function is repeated from the pairwise crossover approach.

We use two forms of genetic programming with the only difference being the initial population - we currently use a set of randomly generated tactics for one form and the generated patterns for the second.

6 Feasibility Tests

One of the concerns about the project was that even using human-directed proofs, a slight change in the order of steps being applied may well lead to a pattern being missed. It was decided that it would be worthwhile to assign each of the rules of Isabelle's HOL library to a class (such as 'classical logic' or 'quantifiers and descriptions') and then look for patterns of classes. This approach seemed likely to give some useful results because of techniques (or perhaps habits) used by both people and automatic provers when looking for a proof. It was suggested that there is a tendency to perform steps from the same class together. For instance, it is often the case that people begin by using all possible rewrite rule to simplify the goal as much as possible. It appears likely that many theorem provers may also use a similar principal for heuristics.

A hand comparison of a number of similar proofs has been carried out. The intention of this was to examine if the similarities in the theorem translated into a similarity in the proof. For the most part, it was found that this was the case, however, the examples studied were simple and it is very possible that two complex theorems proven by two different people may well give very different proofs, even if the statement of the theorems are almost identical. This has led to discussion about whether it would be worthwhile to try to recognize a pattern even if the order of two steps were reversed.

7 Conclusion

The project is progressing within the expected time frame and current results are encouraging.

We have already implemented software which finds patterns from the proof corpus(es) as described above. We have discovered an encouraging number and range of patterns within the proof corpus, however this varies with the particular part of the corpus we use and with the various thresholds we define. We have implemented the feasibility tests described above and have found that results reflect our expectations for these.

We have implemented the evolutionary programming techniques described. These implementations have already provided some results which are under analysis.

We already have some complete tactics from the previous steps and have utilised graphical techniques to allow examination of the statistics. These graphical techniques compare factors such as time (that the evolution step is allowed to run) against the improvement in tactics (at this stage this is measured by the score assigned to the tactic by the scoring function) and the proportion of the corpus explained by the new tactics from each of the evolutionary steps. These graphical models allow us to test different variables and thus refine the weighting and thresholds that are required at various stages of the project.

8 Future Work

As stated, the intention of the project is to allow the automatic formation of tactics to be used in proof planning and automatic theory formation. It is hoped that this project will allow a way to provide tactics which will help guide proof search and will reduce the amount of human intervention needed for theory formation. If successful, this procedure should be able to be integrated into automatic theorem provers and should help reduce the search necessary to find a successful proof. This, in turn, should help improve the success rate of the theorem prover. Theorem provers generally have a maximum time for finding a successful proof so improving the search direction should allow an increase in the number of theorems which can be proved before a timeout occurs.

On a much more localized view, the project would be viewed to be a success if we could demonstrate that significant patterns had been found and that sensible tactics had been formed using these patterns. It would be hoped that tactics could be discovered which made a difference in the search space required to find proofs of a certain type – or even the likelihood of certain types of proof succeeding. Some principles can already be noticed using intuition and common sense, such as the principle that rewrite rules generally occur within a cluster of rewrite rules, and that many proofs begin with the elimination of quantifiers. It would seem a reasonable hope that tactics representing these (or similar) observations could be found.

There are many ways to check the successfulness of the project at various stages. At the final stage it would seem appropriate to evaluate the produced tactics by inspecting them for mathematical "sensibility", i.e., do they make sense within the context of the theorems studied? Do they seem like a sensible approach? A more concrete evaluation would be to enter these new tactics as heuristics within an automatic theorem prover and look for any changes/improvements in its performance.

At a much earlier stage, one suggested evaluation was to see if the patterns found show any expected results, such as existing tactics. This would be expected to occur when using the Isabelle library (for example) as theorems are used as proof steps, if each of these theorems are broken up into their proof traces and so on until the low level HOL logic theorems are reached, then we would expect the proof traces of commonly used lemmas to show up as frequently occurring patterns.

The main step remaining is to test the discovered tactics, this introduces the problem of what we define as a success. There is no reason that our discovered tactics will terminate at the end of the proof so this cannot be used as a criteria. However, it appear that being able to successfully apply each step of a tactic sequentially would itself indicate some measure of success for the tactic.

There is no reason why the technique outlined here could not be used with a number of systems and it would be useful to test how well it applies to different theorem proving systems. In particular, it would be interesting to see how noticeable the difference is between types of tactics discovered from different systems.

Most of the possibilities suggested for further work at this stage involve the amount of data abstracted. It would be interesting to see if other probabilistic techniques such as LLMs could be used to include information about the state of the proof goal when the new tactics should be applied.

References

[**LLM**] A. Berger. The improved iterative scaling algorithm: A gentle introduction, 1997.

[**Bundy**] Alan Bundy. A science of reasoning. in J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178-198, MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[**VLMM**] Aphrodite Galata, Neil Johnson, and David Hogg. Learning variable length Markov models of behaviour. *Computer vision and image understanding: CVIU*, 81(3):398-413,2001.

[**Milner *et al***] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[**Kerber *et al***] M. Jamnik, M. Kerber and M. Pollet. Automatic learning in proof planning. In F. van Harmelen, editor, *Proceedings of 15th ECAI*. European Conference on Artificial Intelligence, 2002.

[**Koza**] J. R. Koza. *Genetic programming: On the programming of computers by means of natural selection*. The MIT Press, 1992.

[**Levine & Humphreys**] John Levine and David Humphreys. Learning action strategies for planning domains using genetic programming.

[**Silver**] B. Silver. *Using meta-level inference to constrain search and to learn strategies in equation solving*. PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1984. Published as a book by North Holland.

[**Newell & Simon**] H. A. Simon and A. Newell. Heuristic problem solving: The next advance in operations research. *Operations research*, 6(1), 1958.

[**Isabelle**] L. C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.

[**Ron *et al***] D. Ron, Y. Singer and N Tishby. The power of amnesia: Learning probabilistic automata with variable memory length, *Machine Learning*,25, 1996.

[**Mizar**] P. Rudnicki. An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs*, Bastad, 1992. Chalmers University of Technology. See http://mizar.org for up-to-date information on Mizar and the Journal of Formalized Mathematics.

CORE and HULL Constructors in Gödel's Class Theory

JOHAN G. F. BELINFANTE AND TIFFANY D. GOBLE [1]

Georgia Institute of Technology, Atlanta, GA 30332–0160 (U.S.A.)
`belinfan@math.gatech.edu`

Abstract

The `GOEDEL` *program, a computer implementation of Gödel's algorithm for class formation in Mathematica^{TM} was used for formulating definitions and discovering theorems about topology and its generalizations, working within Gödel's class theory. A general characterization of CORE[x] and HULL[x] functions discovered in the course of this work is the primary focus of this paper.*

1 Introduction

Computers are not only valuable for automated reasoning and for formal verification in mathematics, but can also contribute significantly to the formulation of definitions, simplifying the statements and the proofs of theorems, finding generalizations, and can sometimes even lead to the discovery of new theorems. This is true in part because the very fact that one is using a computer will suggest natural questions that otherwise may not have been considered, and also in part because computers think in ways that are distinctly non-human. Larry Wos has aptly expressed this in the following words:

> "The human mind will never be replaced, ... but the advantage of computers is their utter lack of preconceptions. They can follow paths that are totally counterintuitive." (Chang, [2004])

Since set theory can be relied upon to formulate practically everything of interest in modern mathematics, it is arguably worthwhile to expend the considerable effort needed to develop a substantial body of standard mathematical facts which can serve as a foundation for automated reasoning involving set theory, building on Robert Boyer's seminal observation that automated reasoning in set theory can be performed within first order logic by using Kurt Gödel's reformulation of the von Neumann-Bernays axioms.

The first author's `GOEDEL` program is used for formal verification and McCune's `Otter` program is used for automatic proof search. The intention is to use the `GOEDEL` program primarily to discover how to formulate definitions and theorems, and to explore what needs to be proved, and then later to go back and find clean proofs of these results using `Otter`. Since the `GOEDEL` program itself does not produce explicit proofs, results obtained with this program will be called derivations rather than proofs, even though such derivations often are in fact more detailed than what passes for a proof in common

[1]This research was supported on NSF ITR contract DMS 0312618.

parlance. A current version of the `GOEDEL` program and a large number of sample notebooks illustrating its use are available on the first author's website:

`http://www.math.gatech.edu/~belinfan/research/`

In the past year, some preliminary successes were achieved by the authors using the `GOEDEL` computer program to derive basic theorems of point-set topology and its generalizations. This work was in part inspired by a remarkable paper by McCune and Wick ([1989]) in which `Otter` was used to prove some theorems of point-set topology before a completely adequate set-theoretic basis was available. Another early effort (Farmer [1991]) to apply automated reasoning to topology produced proofs of two theorems in metric-space topology. Considerable efforts have also been made (Bancerek, [1997]) using `Mizar` to formalize topological notions for the purpose of computer verification. There have undoubtedly been other pioneering efforts to apply computers to reasoning in topology, and it is sincerely hoped that there will be many more in the future.

Unlike `Otter`, the `GOEDEL` program is not an automated reasoning program, and does not produce explicit proofs, but it does contain numerous rewrite rules for simplifying descriptions of classes and assertions about them, and there is a (fairly primitive) construct called `SubstTest` that can be used to carry out deductions by hand. This is one of several tools that permit one to use existing rewrite rules to deduce new rewrite rules, thereby providing a means for the program to gradually evolve into an ever increasingly powerful reasoning assistant. Some theorems about the T1 and T2 separation axioms, the cofinite topology, and compactness were among the results that were derived using `GOEDEL`. Further details about these and other applications to topology can be found on the first author's website. This paper will focus on just one aspect of this ongoing research, a generalization of Kuratowski's characterization of topologies via closure operators that was discovered in the course of our work.

Because our work is being done within the framework of the Gödel-Bernays class theory, the collections of sets that can be considered do not have to be sets. The axiom of regularity and the axiom of choice are not assumed to hold unless explicitly mentioned. Generalizing topological ideas to proper classes is not just an idle pastime. In ordinal number theory, for example, an important concept is that of a full (or transitive) class. A class is *full* if all its members are subsets. The class `FULL` of all full sets is technically not a topology because it is a proper class, but it shares some properties of a topology. In particular, the class of all full sets is closed under arbitrary unions, and under arbitrary intersections, too. The class `H[FINITE]` of hereditarily finite sets, for example, can be characterized as the interior of the class `FINITE` of finite sets with respect to this pseudo-topology.

The needs of automated reasoning raise many mathematical questions that are not readily answered in the standard mathematical literature. This is particularly true in situations where the standard literature deals only with the case of sets and is silent on the issue whether sethood is really necessary. Some of the issues requiring attention are just details that can be readily resolved. For example, upon proving a standard result of the form `A` $\Rightarrow$ `B`, one might wish to add a rewrite rule that automatically rewrites `A` to `B`. Doing so would only be justified if the converse implication also holds, forcing one therefore to think about whether the converse is true even in situations where the converse statement might not otherwise be particularly interesting. Sometimes, however, genuinely interesting issues are brought to one's attention in this fashion. In the course of the research reported in this paper, a number of interesting mathematical questions arose for which the authors were unable to provide entirely satisfactory resolutions. Some of these open questions will be mentioned as we go along.

2 Notation

In the Gödel class theory, equality and the membership predicate $\in$ are taken as primitive undefined concepts, subject to various axioms, and everything is a class. By definition, a class is a *set* if there is a class to which it belongs. Classes that are not sets are called *proper classes*. The argument leading to Russell's paradox can be used to show that Russell's class of all sets that do not belong to themselves is a proper class. The axiom of replacement implies that any subclass of a set is a set, and from this one can deduce that the class V of all sets must also be a proper class. Another familiar example of a proper class is the class Ω of all ordinals.

Sets are sometimes called *small* classes, the idea being that all the known examples of proper classes are extremely large. This notion is reinforced by the axiom of replacement which implies that any subclass of a set is a set. But it is somewhat embarrassing that one can not even show for example that every proper class contains an infinite subset. This open question came up in the course of dealing with theorems about finite topological spaces. The natural question arose whether one could simply add a rewrite rule that transforms the statement $\mathtt{P}[\mathtt{x}] \subset \mathtt{FINITE}$ to $\mathtt{x} \in \mathtt{FINITE}$. These statements are of course trivially equivalent when x is a set because any set belongs to its power set, but to avoid a conditional rewrite rule, one really wants to know whether sethood is needed. The general case amounts to the question whether a proper class must always contain an infinite subset. (A similar question, whether an infinite set must contain a countable subset, is known to require an application of the axiom of choice.)

Because the Zermelo-Fraenkel version of set theory is the one most familiar to most mathematicians, proper classes are used rather sparingly in the literature. Even authors (Rubin [1967], Mendelson [1987]) that do embrace the NBG axioms for class theory generally mention proper classes only to convince the reader that the paradoxes of naive set theory are resolved, but rarely take full advantage of constructions involving proper classes. Since Gödel's algorithm (Gödel, [1940]) routinely produces such constructions, proper classes feature prominently in our work. For this reason it is appropriate to review some basic proper classes that will be used in the sequel.

In addition to the proper class V, there is another proper class, the membership relation, whose existence is postulated by one of the axioms in Gödel's class theory. This axiom asserts that there is a class E whose members are all ordered pairs $\langle \mathtt{x}, \mathtt{y} \rangle$ satisfying $\mathtt{x} \in \mathtt{y}$.

The class image[x,y] is defined as the range of the restriction of the relation x to the class y. The *vertical section* of a relation x at a set y is $\mathtt{image}[\mathtt{x}, \{\mathtt{y}\}]$, where $\{\mathtt{y}\}$ denotes the singleton of y. A relation x is *thin* if image[x,y] is a set whenever y is set. Any set, of course, is thin, and the axiom of replacement is equivalent to the assertion that all functions are thin. It can be shown that x is thin if all its vertical sections are sets.

The class ub[x,y] of upper bounds of a class y with respect to a relation x is defined to be the class

$$\mathtt{ub}[\mathtt{x},\mathtt{y}] = \mathtt{image}[\mathtt{x}',\mathtt{y}]\,',$$

where $\mathtt{x}'$ denotes the complement of the class x. Similarly, the class lb[x,y] of lower bounds is defined by replacing x with its inverse:

$$\mathtt{lb}[\mathtt{x},\mathtt{y}] = \mathtt{ub}[\mathtt{inverse}[\mathtt{x}],\mathtt{y}].$$

An important application of this is the formula for the intersection of a collection of sets,

$$\mathtt{lb}[\mathtt{E},\mathtt{x}] = \bigcap \mathtt{x}.$$

The sum class $\bigcup \mathtt{x}$ and the power class P[x] of a class x can be defined in terms of the membership relation by the formulas

$$\bigcup \mathtt{x} = \mathtt{image}[\mathtt{inverse}[\mathtt{E}],\mathtt{x}]$$

and

$$\mathtt{P}[\mathtt{x}] = \mathtt{image}[\mathtt{E}',\mathtt{x}]\,'.$$

According to the sum class and power set axioms, these are sets when x is a set. It follows from the sum class axiom that the inverse of the membership relation is thin.

For any class x, the upper bound relation

$$\mathtt{UB}[\mathtt{x}] = (\mathtt{x}' \circ \mathtt{inverse}[\mathtt{E}])\,' \cap (\mathtt{V} \times \mathtt{V})$$

is the class of ordered pairs $\langle \mathtt{y},\mathtt{z}\rangle$ such that z = ub[x,y]. The lower bound relation is defined by LB[x] = UB[inverse[x]]. In particular, the subset relation S = UB[E] is the class of ordered pairs $\langle \mathtt{y},\mathtt{z}\rangle$ such that $\mathtt{y} \subset \mathtt{z}$. It follows from the power set axiom that the inverse of the subset relation is thin.

The identity relation is $\mathtt{Id} = \mathtt{S} \cap \mathtt{inverse}[\mathtt{S}]$. The restriction of the identity relation to a class x is denoted by $\mathtt{id}[\mathtt{x}]$. The class fix[x] of all fixed points of x, defined by

$$\mathtt{fix}[\mathtt{x}] = \{\mathtt{y} \mid \langle \mathtt{y},\mathtt{y}\rangle \in \mathtt{x}\},$$

is related to the identity relation by

$$\mathtt{x} \cap \mathtt{Id} = \mathtt{id}[\mathtt{fix}[\mathtt{x}]].$$

3 Eliminating set variables

All statements in mathematics can be automatically converted by means of Gödel's algorithm into equations without set-variables, something which Alfred Tarski and Steven Givant ([1987]) had shown could be done in theory, on the basis of a calculus of relations. This result is not limited however to the special formalism that they consider, but can also be achieved in the more traditional setting of NBG class theory. When this process (called assert in the GOEDEL program) is applied to a statement containing quantifiers over set-variables, the statement is converted into a logically equivalent equation without quantifiers. In the current version of the GOEDEL program, the equations that one gets are often subsequently converted by rewrite rules to simpler statements of a non-equational nature. For example, the axiom of regularity is transformed into the statement that the universal class is the only class that contains its own power class. It should be noted that this technique cannot be used to eliminate quantifiers over class variables.

A recent major discovery provides another method for eliminating set variables, using a process that has been named reification (Belinfante, [2003]). The idea is to associate to each constructor `f[x]` in the Gödel class theory the relation `R` of all ordered pairs $\langle \mathtt{x}, \mathtt{y} \rangle$ such that `y` belongs to the class `f[x]`,

$$\mathtt{R} = \mathtt{reify}[\mathtt{x}, \mathtt{f}[\mathtt{x}]] = \{\langle \mathtt{x}, \mathtt{y} \rangle \mid \mathtt{y} \in \mathtt{f}[\mathtt{x}]\}.$$

For each constructor `f`, there is a formula expressing the reification of composite constructors `f[g[x]]` in terms of the reification of the inner constructor `g`. These reification rules can often serve as a substitute for Gödel's algorithm for eliminating set variables, with improved execution time and cleaner output. Some examples that illustrate this technique will be presented in this paper.

In many cases the result of eliminating variables produces formulas involving function constructions. To each relation `x` there corresponds a function `VERTSECT[x]` which assigns to each set `y` the vertical section of `x` at `y` whenever the vertical section is a set. This important constructor can be used to define many important special functions. For example, the function `POWER = VERTSECT[inverse[S]]` assigns to each set its power set, the function `SINGLETON = VERTSECT[Id]` assigns to each set its singleton, and the function

$$\mathtt{BIGCAP} = \mathtt{VERTSECT}[\mathtt{LB}[\mathtt{E}]]$$

takes any nonempty collection of sets `x` to its intersection $\bigcap \mathtt{x}$. A relation `x` is thin if `domain[VERTSECT[x]] = V`. The relation `LB[E]` fails to be thin because its vertical section at the empty set is the proper class `V`; the domain of `BIGCAP` is the class $\{\mathtt{0}\}'$ of nonempty sets. In principle, any function `f` can be expressed as a `VERTSECT`,

$$\mathtt{f} = \mathtt{VERTSECT}[(\mathtt{inverse}[\mathtt{E}] \circ \mathtt{f}) \cup (\mathtt{domain}[\mathtt{f}]' \times \mathtt{V})].$$

This permits one to construct `f` whenever formulas for $\mathtt{inverse}[\mathtt{E}] \circ \mathtt{f}$ and `domain[f]` are available.

A closely related constructor for functions is

$$\mathtt{IMAGE}[\mathtt{x}] = \mathtt{VERTSECT}[\mathtt{x} \circ \mathtt{inverse}[\mathtt{E}]],$$

which assigns to each set `y` the image `image[x,y]` provided the latter is a set. For example, the function `BIGCUP = IMAGE[inverse[E]]` assigns to each set its sum class, the function `IMAGE[id[x]]` assigns to each set its intersection with `x`, and the function `IMAGE[SWAP]` takes `x` to `inverse[x]`. For this last function, it is generally more convenient to work with its restriction to the class $\mathtt{P}[\mathtt{V} \times \mathtt{V}]$ of all small relations,

$$\mathtt{INVERSE} = \mathtt{IMAGE}[\mathtt{SWAP}] \circ \mathtt{id}[\mathtt{P}[\mathtt{V} \times \mathtt{V}]],$$

because this restriction is one-to-one; indeed, this restriction is an involution, that is, a function which is equal to its own inverse,

$$\mathtt{inverse}[\mathtt{INVERSE}] = \mathtt{INVERSE}.$$

In a later section, an application of this function to integer arithmetic will be described.

The domain of `IMAGE[x]` is

$$\mathtt{domain}[\mathtt{IMAGE}[\mathtt{x}]] = \mathtt{P}[\mathtt{domain}[\mathtt{VERTSECT}[\mathtt{x}]]].$$

In particular, when `x` is thin, the domains of both `VERTSECT[x]` and `IMAGE[x]` are equal to the universal class `V`. One of the most-used rewrite rules in the `GOEDEL` program transforms $\mathtt{x} \circ \mathtt{inverse}[\mathtt{E}]$ to $\mathtt{inverse}[\mathtt{E}] \circ \mathtt{IMAGE}[\mathtt{x}]$ whenever `x` is thin. This rewrite rule in effect automatically replaces thin relations with functions.

While any function can be written as `VERTSECT[y]`, not all functions can be written in the form `IMAGE[y]`. For the special case that `y` is thin, the `GOEDEL` program yields a simple characterization of such functions. In one direction, one has

$$(\mathtt{x} = \mathtt{IMAGE}[\mathtt{y}] \ \& \ \mathtt{thin}[\mathtt{y}]) \ \Rightarrow \ \mathtt{BIGCUP} \circ \mathtt{IMAGE}[\mathtt{x}] = \mathtt{x} \circ \mathtt{BIGCUP}.$$

Conversely, if $\mathtt{BIGCUP} \circ \mathtt{IMAGE}[\mathtt{x}] = \mathtt{x} \circ \mathtt{BIGCUP}$, and $\mathtt{x} \subset \mathtt{V} \times \mathtt{V}$, then $\mathtt{x} = \mathtt{IMAGE}[\mathtt{y}]$, where $\mathtt{y} = \mathtt{inverse}[\mathtt{E}] \circ \mathtt{x} \circ \mathtt{SINGLETON}$ and `y` is thin.

4 Definitions of core and hull

The class `core[x,y]` is defined to be the union of all sets that belong to `x` and are contained in `y`.

$$\mathtt{core}[\mathtt{x}, \mathtt{y}] = \bigcup (\mathtt{x} \cap \mathtt{P}[\mathtt{y}]).$$

If `x` is the collection of open sets for a topological space, and if `y` is a subset of the space, then `core[x,y]` is the interior of `y`. This concept is also of interest when `x` and `y` are proper classes. For example the class of hereditarily finite sets is `core[FULL,FINITE]`. In general, the class `H[x] = core[FULL,x]` is the largest full subclass of `x`.

The class `hull[x,y]` is defined to be the intersection of all sets that belong to `x` and which contain `y`.

$$\mathtt{hull}[\mathtt{x}, \mathtt{y}] = \bigcap (\mathtt{x} \cap \mathtt{image}[\mathtt{S}, \{\mathtt{y}\}]).$$

For example, if `x` is the collection of closed sets of a topological space, and if `y` is a subset of the space, then `hull[x,y]` is the closure of `y`.

The formal similarity between the definitions of `core` and `hull` breaks down when proper classes are considered. Since there are no sets that contain a proper class, one has `hull[x,y] = V` whenever `y` is a proper class. This circumstance has practical repercussions. For example, for every class `x` there is a smallest class `tc[x]` which is full and contains `x`. When `x` is a set, the transitive closure is given by the simple formula `tc[x] = hull[FULL,x]`, but when `x` is a proper class, a slightly different construction is required, based on the intuitive notion that a proper class can be approximated in some sense by very large subsets. To make this vague idea more precise, some additional definitions will be needed, which will now be explained.

A function `f` is *idempotent* if $\mathtt{f} \circ \mathtt{f} = \mathtt{f}$. If a function is idempotent, then every element of its range is a fixed point: `range[f] = fix[f]`. It is convenient to introduce two families of idempotent functions `CORE[x]` and `HULL[x]` each depending on a parameter `x` which in principle can be any class. The function `CORE[x]` can be formally characterized as the class of all ordered pairs $\langle \mathtt{y}, \mathtt{z} \rangle$ such that `z = core[x,y]` and the function `HULL[x]` is the class of pairs $\langle \mathtt{y}, \mathtt{z} \rangle$ such that `z = hull[x,y]`. If `x` is the collection of open sets for a topological space, then the restriction of `CORE[x]` to the class of all subsets of the topological space is the interior operator. If `x` is the collection of closed sets for a topological space, then the restriction of `HULL[x]` to the class of all subsets of the topological space is the closure operator. Kuratowski characterized these closure operators and showed that a topology is uniquely determined by its closure operator.

Despite the similarity of the characterizations of the functions `CORE[x]` and `HULL[x]`, some of their properties are actually quite different, and this already shows up in the formulas used to define them. Since class formation is part of the metatheory of Gödel's class theory, and not part of the theory itself, the similarity of the characterizations of these functions in terms of requirements for ordered pairs to belong to them is somewhat misleading. Gödel's algorithm yields an equational definition for each of these functions. In the case of `CORE[x]` one obtains

$$\mathtt{CORE}[\mathtt{x}] = \mathtt{BIGCUP} \circ \mathtt{IMAGE}[\mathtt{id}[\mathtt{x}]] \circ \mathtt{POWER}.$$

For `HULL[x]` one finds a rather different formula, namely,

$$\mathtt{HULL}[\mathtt{x}] = \mathtt{VERTSECT}[(\mathtt{inverse}[\mathtt{E}]' \circ \mathtt{id}[\mathtt{x}] \circ \mathtt{S})'].$$

It is probably worth pointing out that this formula does not actually appear explicitly in the `GOEDEL` program, the reason being that from this definition one can derive the formula

$$\mathtt{inverse}[\mathtt{E}]' \circ \mathtt{id}[\mathtt{x}] \circ \mathtt{S} = \mathtt{inverse}[\mathtt{E}]' \circ \mathtt{HULL}[\mathtt{x}].$$

The latter formula occurs as a rewrite rule that subsumes the preceding formula. A similar result holds, by the way, for `CORE[x]`,

$$\mathtt{inverse}[\mathtt{E}] \circ \mathtt{id}[\mathtt{x}] \circ \mathtt{inverse}[\mathtt{S}] = \mathtt{inverse}[\mathtt{E}] \circ \mathtt{CORE}[\mathtt{x}].$$

Further dissimilarities are found in other properties of these functions. Consider for example the domains of these functions. When `y` is a set, so is its power set `P[y]`. Since any subclass of a set is a set, the intersection $\mathtt{x} \cap \mathtt{P}[\mathtt{y}]$ is a set, and hence, by the sum class axiom, $\mathtt{z} = \mathtt{core}[\mathtt{x}, \mathtt{y}] = \bigcup(\mathtt{x} \cap \mathtt{P}[\mathtt{y}])$ is a set. Consequently, the domain of `CORE[x]` is the class `V` of all sets. On the other hand, the intersection of a collection of sets is a set if and only if that collection is not empty. Consequently, the domain of `HULL[x]` is the class of all subsets of members of `x`,

$$\mathtt{domain}[\mathtt{HULL}[\mathtt{x}]] = \mathtt{image}[\mathtt{inverse}[\mathtt{S}], \mathtt{x}].$$

By the axiom of replacement, it follows from this that while the function `CORE[x]` is always a proper class, the function `HULL[x]` is a set if and only if `x` is a set.

The function `TC = HULL[FULL]` provides a method to construct the transitive closure of any class,

$$\mathtt{tc}[\mathtt{x}] = \bigcup \mathtt{image}[\mathtt{TC}, \mathtt{P}[\mathtt{x}]].$$

In practice one also needs an additional formula,

$$\mathtt{tc}[\mathtt{x}] = \mathtt{range}[\mathtt{iterate}[\mathtt{inverse}[\mathtt{E}], \mathtt{x}]],$$

which allows one to use induction to derive the properties of the transitive closure. Both of these formulas hold for any class x, not just for sets.

Another important example is the family of functions

$$\mathtt{ADJOIN}[\mathtt{x}] = \mathtt{HULL}[\mathtt{image}[\mathtt{S}, \{\mathtt{x}\}]].$$

If x is a set, this is a total function which takes any set y to the set $\mathtt{x} \cup \mathtt{y}$. When x is not a set, the function is the empty set.

In general, the relation iterate[x, y] can be characterized (Belinfante [2003]) by the conditions that its vertical section at the empty set is the class y, and for each natural number n, the image under x of the vertical section at n produces the vertical section at the successor of n. Explicitly, the following uniqueness theorem holds for iteration:

$$\mathtt{z} \circ \mathtt{SUCC} = \mathtt{x} \circ \mathtt{z} \ \ \& \ \ \mathtt{image}[\mathtt{z}, \{0\}] = \mathtt{y} \ \Rightarrow \ \mathtt{iterate}[\mathtt{x}, \mathtt{y}] = \mathtt{z} \circ \mathtt{id}[\omega],$$

where $\omega = \{0, 1, 2, \dots\}$ denotes the set of all natural numbers, and SUCC denotes the successor function, which takes any set x to its successor $\mathtt{succ}[\mathtt{x}] = \mathtt{x} \cup \{\mathtt{x}\}$.

5 Open questions concerning Uclosure and Aclosure

The Uclosure of any class x is the class of all unions of subsets of x,

$$\mathtt{Uclosure}[\mathtt{x}] = \mathtt{image}[\mathtt{BIGCUP}, \mathtt{P}[\mathtt{x}]].$$

A familiar application of this is the construction of a topology from a topological base. Similarly, Aclosure[x] is the class of all intersections of subsets of x,

$$\mathtt{Aclosure}[\mathtt{x}] = \mathtt{image}[\mathtt{BIGCAP}, \mathtt{P}[\mathtt{x}]].$$

Every class x is contained in its own Aclosure and Uclosure. A class x is closed under arbitrary intersections if it satisfies Aclosure[x] = x, and is closed under arbitrary unions if Uclosure[x] = x. This is the case, for example, for the class invar[x] of all sets invariant under an operation x,

$$\mathtt{invar}[\mathtt{x}] = \{\mathtt{y} \mid \mathtt{image}[\mathtt{x}, \mathtt{y}] \subset \mathtt{y}\}.$$

Important examples include the class FULL = invar[inverse[E]], and any power class $\mathtt{P}[\mathtt{x}] = \mathtt{invar}[\mathtt{x}' \times \mathtt{V}]$. Further examples can be constructed by using formulas such as

$$\mathtt{invar}[\mathtt{x} \cup \mathtt{y}] = \mathtt{invar}[\mathtt{x}] \cap \mathtt{invar}[\mathtt{y}].$$

The idempotent functions UCLOSURE and ACLOSURE take a set x to Uclosure[x] and ACLOSURE, respectively:

$$\mathtt{ACLOSURE} = \mathtt{IMAGE}[\mathtt{BIGCAP}] \circ \mathtt{POWER},$$
$$\mathtt{UCLOSURE} = \mathtt{IMAGE}[\mathtt{BIGCUP}] \circ \mathtt{POWER}.$$

Each of these functions is a HULL function:

$$\mathtt{ACLOSURE} = \mathtt{HULL}[\mathtt{fix}[\mathtt{ACLOSURE}]],$$
$$\mathtt{UCLOSURE} = \mathtt{HULL}[\mathtt{fix}[\mathtt{UCLOSURE}]].$$

It is currently not known whether the functions UCLOSURE and ACLOSURE commute. At issue here is whether the distributive law extends to infinite unions and intersections; can an infinite union of infinite intersections be rewritten as an infinite intersection of infinite unions, and vice versa? A closely related question is whether `fix[ACLOSURE]` is invariant under UCLOSURE, and conversely.

Since the functions `CORE[x]` and `HULL[x]` are both idempotent, their ranges and fixed point classes are equal. In the case of `CORE[x]`, one has

$$\mathtt{Uclosure}[\mathtt{x}] = \mathtt{range}[\mathtt{CORE}[\mathtt{x}]] = \mathtt{fix}[\mathtt{CORE}[\mathtt{x}]],$$

but for the case of `HULL[x]` all that has been proved at this point is that

$$\mathtt{Aclosure}[\mathtt{x}] \subset \mathtt{range}[\mathtt{HULL}[\mathtt{x}]] = \mathtt{fix}[\mathtt{HULL}[\mathtt{x}]].$$

It remains an open question whether the class `fix[HULL[x]]` is in fact always equal to the class `Aclosure[x]`. Equality has been proved for the important special case that `x` is a set, and also for various special proper classes. It would be desirable to have either a general proof that these classes are always equal or else a counterexample if they are not always equal. Another open quesion is whether the constructor `Aclosure` is idempotent in general, as is known to be the case for `Uclosure`. If `x` is a set, one has

$$\mathtt{Aclosure}[\mathtt{Aclosure}[\mathtt{x}]] = \mathtt{Aclosure}[\mathtt{x}],$$

which suffices to show that the function ACLOSURE is idempotent. Incidentally, the closely related operation `fix[HULL[x]]` is known to be idempotent for arbitrary classes; in fact,

$$\mathtt{HULL}[\mathtt{fix}[\mathtt{HULL}[\mathtt{x}]]] = \mathtt{HULL}[\mathtt{x}].$$

The functions `CORE[x]` and `HULL[x]` satisfy the equations

$$\mathtt{CORE}[\mathtt{Uclosure}[\mathtt{x}]] = \mathtt{CORE}[\mathtt{x}],$$
$$\mathtt{HULL}[\mathtt{Aclosure}[\mathtt{x}]] = \mathtt{HULL}[\mathtt{x}].$$

Thus, for example, it makes no difference if one replaces a topology by a base for the topology in defining interiors.

One of the more fascinating properties of the function UCLOSURE is that it commutes with `IMAGE[IMAGE[x]]` for any class `x`. A special case of this was used in the study of relative topologies.

6 Characterizing CORE and HULL

It is convenient to express various strong versions of monotonicity in terms of the subset relation `S`. A function `f` is *monotone* if

$$\mathtt{f} \circ \mathtt{S} \circ \mathtt{inverse}[\mathtt{f}] \subset \mathtt{S}.$$

This quantifier-free statement is equivalent to the condition

$$\forall \mathtt{u}, \mathtt{v}, \mathtt{x}, \mathtt{y}\ ((\langle \mathtt{u}, \mathtt{x}\rangle \in \mathtt{f}\ \ \&\ \ \langle \mathtt{v}, \mathtt{y}\rangle \in \mathtt{f}\ \ \&\ \ \mathtt{u} \subset \mathtt{v}) \Rightarrow \mathtt{x} \subset \mathtt{y}).$$

Similarly, a function is *antitone* if

$$\mathtt{f} \circ \mathtt{inverse}[\mathtt{S}] \circ \mathtt{inverse}[\mathtt{f}] \subset \mathtt{S}.$$

A function is *total* if `domain[f] = V`. A class `x` is *hereditary* if every subset of a member is a member, that is, if `image[inverse[S], x] = x]`. It will be said that `x` and `y` *subcommute* if $\mathtt{x}\circ\mathtt{y} \subset \mathtt{y}\circ\mathtt{x}$. Any relation that subcommutes with `S` has a hereditary domain. Any monotone function with a hereditary domain subcommutes with the subset relation, and conversely, any function that subcommutes with `S` is monotone. In other words, the condition that a function `f` subcommutes with `S` is equivalent to saying that it is not only monotone, but its domain is hereditary. A total monotone function `f` satisfies the even stronger condition that $\mathtt{S} \subset \mathtt{inverse}[\mathtt{f}] \circ \mathtt{S} \circ \mathtt{f}$, and conversely, this condition implies that `f` is monotone and total. A total function which subcommutes with the subset relation also subcommutes with the inverse of the subset relation.

For example, the domain of the monotone function `IMAGE[x]` is a power class,

$$\mathtt{domain}[\mathtt{IMAGE}[\mathtt{x}]] = \mathtt{P}[\mathtt{domain}[\mathtt{VERTSECT}[\mathtt{x}]]],$$

and is therefore hereditary. Consequently `IMAGE[x]` subcommutes with `S`. The function `IMAGE[x]` is total when `x` is thin; `IMAGE[x]` subcommutes with `inverse[S]` if and only if `x` is thin. This fact had been proved in one direction using `Otter`. The discovery that the converse holds was motivated by the desire to avoid a conditional rewrite rule in the `GOEDEL` program; conditional rewrite rules slow the program down significantly. Some additional facts did require adding conditional rewrite rules: If `x` is a function, then `IMAGE[x]` commutes with `S`, and if `x` is a total one-to-one function, then `IMAGE[x]` commutes with `S`.

The idempotent functions `CORE[x]` and `HULL[x]` are both monotone, and both of them subcommute with the subset relation because their domains are hereditary. The function `CORE[x]`, of course, has the stronger property of being total, and therefore subcommutes also with `inverse[S]`, whereas this is generally not the case for `HULL[x]`. The other important difference between these functions is that `HULL[x]` is contained in `S`, whereas `CORE[x]` is contained in `inverse[S]`.

These properties characterize the functions `CORE[x]` and `HULL[x]`. If a monotone idempotent function has a hereditary domain and is contained in the subset relation, then it is a `HULL` function,

$$(\mathtt{f} \circ \mathtt{f} = \mathtt{f}\ \ \&\ \ \mathtt{FUNCTION}[\mathtt{f}]\ \ \&\ \ \mathtt{f} \subset \mathtt{S}\ \ \&\ \ \mathtt{f} \circ \mathtt{S} \subset \mathtt{S} \circ \mathtt{f}) \Rightarrow \mathtt{f} = \mathtt{HULL}[\mathtt{fix}[\mathtt{f}]].$$

If a total monotone idempotent function is contained in the inverse of the subset relation, then it is a `CORE` function,

$$\begin{aligned}(&\mathtt{f} \circ \mathtt{f} = \mathtt{f}\ \ \&\ \ \mathtt{FUNCTION}[\mathtt{f}]\ \ \&\ \ \mathtt{domain}[\mathtt{f}] = \mathtt{V} \\ &\ \ \&\ \mathtt{f} \circ \mathtt{S} \subset \mathtt{S} \circ \mathtt{f}\ \ \&\ \ \mathtt{f} \subset \mathtt{inverse}[\mathtt{S}]) \quad \Rightarrow \quad \mathtt{f} = \mathtt{CORE}[\mathtt{fix}[\mathtt{f}]].\end{aligned}$$

The interiors and closures of subsets of a topological space are related via relative complementation. The result does not depend on the topology axioms. A general result can be derived using the above characterization of HULL functions. For any set x there is a relative complementation function RC[x] consisting of all ordered pairs $\langle \mathtt{y}, \mathtt{z} \rangle$ such that $\mathtt{y} \cup \mathtt{z} = \mathtt{x}$ and $\mathtt{y} \cap \mathtt{z} = \mathtt{0}$.

$$\mathtt{RC}[\mathtt{x}] = \mathtt{DISJOINT} \cap \mathtt{image}[\mathtt{inverse}[\mathtt{CUP}], \{\mathtt{x}\}].$$

This function is antitone, and is its own inverse. Note that RC[x] = 0 when x is a proper class. Since the function CORE[y] is monotone, the composite function $\mathtt{RC}[\mathtt{x}] \circ \mathtt{CORE}[\mathtt{z}] \circ \mathtt{RC}[\mathtt{x}]$ is monotone. The other conditions in the characterization of a HULL function also hold, and so, applying the characterization of HULL to this special case yields the formula

$$\mathtt{RC}[\mathtt{x}] \circ \mathtt{CORE}[\mathtt{y}] \circ \mathtt{RC}[\mathtt{x}] = \mathtt{HULL}[\mathtt{image}[\mathtt{RC}[\mathtt{x}], \mathtt{Uclosure}[\mathtt{y}]]].$$

Actually, a slightly more general result holds:

$$\mathtt{HULL}[\mathtt{image}[\mathtt{RC}[\mathtt{x}], \mathtt{y}]] = \mathtt{RC}[\mathtt{x}] \circ \mathtt{CORE}[\mathtt{y}] \circ \mathtt{id}[\mathtt{image}[\mathtt{S}, \mathtt{y}]] \circ \mathtt{RC}[\mathtt{x}].$$

The extra factor id[image[S,y]] is not needed when $\mathtt{0} \in \mathtt{y}$. For the record, we note that this more general formula had in fact been derived by an application of reification before the characterization of HULL had been established.

7 Applications to Topology

A *topology* is a set t of sets which is closed under binary intersections and arbitrary unions, that is, a set satisfying

$$\begin{aligned} \mathtt{t} &= \mathtt{Uclosure}[\mathtt{t}], \\ \mathtt{t} &= \mathtt{image}[\mathtt{CAP}, \mathtt{t} \times \mathtt{t}]. \end{aligned}$$

Any power set is a topology, as are all successor ordinals. In general, the Uclosure of an ordinal number is the successor of its sum class. The *cofinite topology* for any set x holds the empty set, and all subsets of x whose relative complement in x is finite,

$$\mathtt{Uclosure}[\mathtt{image}[\mathtt{RC}[\mathtt{x}], \mathtt{FINITE}]] = \{\mathtt{0}\} \cup \mathtt{image}[\mathtt{RC}[\mathtt{x}], \mathtt{FINITE}].$$

In general, the class binclosed[x] of sets closed under a binary operation x is

$$\mathtt{binclosed}[\mathtt{x}] = \{\mathtt{t} \mid \mathtt{image}[\mathtt{x}, \mathtt{t} \times \mathtt{t}] \subset \mathtt{t}\} = \mathtt{fix}[\mathtt{S} \circ \mathtt{IMAGE}[\mathtt{x}] \circ \mathtt{CART} \circ \mathtt{DUP}].$$

Using this notation, one can write the class TOPS of all topologies as the intersection

$$\mathtt{TOPS} = \mathtt{fix}[\mathtt{UCLOSURE}] \cap \mathtt{binclosed}[\mathtt{CAP}].$$

Note the resemblance of binclosed[x] with invar[x]. In fact, invar[x] can even be written as $\mathtt{binclosed}[\mathtt{x} \circ \mathtt{inverse}[\mathtt{DUP}]]$. It should therefore not come as much of a surprise that binclosed shares some of the properties of invar. In particular, it is closed under arbitrary intersections: Aclosure[binclosed[x]] = binclosed[x].

The class TOPS is closed under arbitrary intersections: Aclosure[TOPS]=TOPS. Any set x generates a smallest topology hull[TOPS, x] that contains x. Since the class binclosed[CAP] is invariant under UCLOSURE, it follows from the characterization of HULL functions that

$$\texttt{HULL}[\texttt{TOPS}] = \texttt{UCLOSURE} \circ \texttt{HULL}[\texttt{binclosed}[\texttt{CAP}]].$$

In other words, the topology generated by a set can be obtained in two steps; first one generates a topological base, and then one applies Uclosure to obtain the topology itself,

$$\texttt{hull}[\texttt{TOPS}, \texttt{x}] = \texttt{Uclosure}[\texttt{hull}[\texttt{binclosed}[\texttt{CAP}], \texttt{x}]].$$

Another corollary is a succinct formula TOPS = image[UCLOSURE, binclosed[CAP]] for the class of all topologies.

If t is a topology, its members are called the open subsets of the topological space $\bigcup \texttt{t}$, and their relative complements are called the closed sets. So $\texttt{c} = \texttt{image}[\texttt{RC}[\bigcup \texttt{t}], \texttt{t}]$ is the set of all closed sets. The interior of a subset $\texttt{x} \subset \bigcup \texttt{t}$ is core[t, x], and its closure is hull[c, x].

If t is any topology, and x is any class, then the set

$$\texttt{image}[\texttt{IMAGE}[\texttt{id}[\texttt{x}]], \texttt{t}] = \{\texttt{z} \mid (\exists \texttt{y} \in \texttt{t})\ \texttt{z} = \texttt{x} \cap \texttt{y}\}$$

is also a topology. When $\texttt{x} \subset \bigcup \texttt{t}$ this is known as the *relative topology* on the subset x. For each open set $\texttt{y} \in \texttt{t}$, the intersection $\texttt{z} = \texttt{x} \cap \texttt{y}$ is open in the relative topology. Since the variable t refers to a set, one can eliminate this variable, and recast the fact that the class of topologies is invariant under the process of forming relative topologies succinctly as follows:

$$\texttt{image}[\texttt{IMAGE}[\texttt{IMAGE}[\texttt{id}[\texttt{x}]]], \texttt{TOPS}] \subset \texttt{TOPS}.$$

Carrying this process of variable-elimination to extremes, one could specialize the above statement to the case that x is any set, and then use reification to eliminate even this one remaining variable, yielding a completely variable-free statement,

$$\texttt{image}[\texttt{IMAGE}[\texttt{CAP}], \texttt{image}[\texttt{CART}, \texttt{range}[\texttt{SINGLETON}] \times \texttt{TOPS}]] \subset \texttt{TOPS}.$$

This amounts to the assertion that TOPS is invariant under the relation obtained by forming the union of the functions IMAGE[IMAGE[id[x]]]. This is not an ordinary union because these functions are all proper classes, but one can nonetheless use the reification rules to compute such nonstandard unions. For any class constructor f[x], one can compute the union of all the classes f[x] for which x is a set as follows:

$$\{\texttt{w} \mid \exists \texttt{x}\ \texttt{w} \in \texttt{f}[\texttt{x}]\} = \texttt{range}[\texttt{reify}[\texttt{x}, \texttt{f}[\texttt{x}]]].$$

In the present case, one needs to use the reification rules for the IMAGE and id constructors:

$$\begin{aligned}&\texttt{reify}[\texttt{x}, \texttt{IMAGE}[\texttt{y}]] = \texttt{SWAP} \circ \texttt{inverse}[\texttt{rotate}[\\ &\texttt{IMAGE}[\texttt{rotate}[\texttt{inverse}[\texttt{reify}[\texttt{x}, \texttt{y}]]]] \circ \texttt{CART} \circ \texttt{SWAP}]] \circ \texttt{SINGLETON},\end{aligned}$$

$$\texttt{reify}[\texttt{x}, \texttt{id}[\texttt{y}]] = \texttt{DUP} \circ \texttt{reify}[\texttt{x}, \texttt{y}].$$

In this way one readily discovers that the class TOPS is invariant under the relation

$$\begin{aligned}&\mathtt{range}[\mathtt{reify}[\mathtt{x}, \mathtt{IMAGE}[\mathtt{IMAGE}[\mathtt{id}[\mathtt{x}]]]]] = \\ &\mathtt{IMAGE}[\mathtt{CAP}] \circ \mathtt{CART} \circ \mathtt{id}[\mathtt{range}[\mathtt{SINGLETON}] \times \mathtt{V}] \circ \mathtt{inverse}[\mathtt{SECOND}].\end{aligned}$$

Before leaving this topic, it is probably worth pointing out that even more is true; the function id[x] can be replaced by any one-to-one function. The class fix[UCLOSURE] is invariant under IMAGE[IMAGE[x]] for any class x, and the class binclosed[CAP] is invariant under IMAGE[IMAGE[x]] when x is any one-to-one function. From this it readily follows that

$$\mathtt{ONEONE}[\mathtt{x}] \Rightarrow \mathtt{image}[\mathtt{IMAGE}[\mathtt{IMAGE}[\mathtt{x}]], \mathtt{TOPS}] \subset \mathtt{TOPS}.$$

8 Transitive closures of relations

An important application of HULL functions is the theory of transitive closures of relations. This relational transitive closure trv[x] should not be confused with the class tc[x] discussed earlier.

A relation $\mathtt{x} \subset \mathtt{V} \times \mathtt{V}$ is *transitive* if $\mathtt{x} \circ \mathtt{x} \subset \mathtt{x}$. The class of all small transitive relations is denoted by TRV. The (relational) *transitive closure* trv[x] of a relation x is the smallest transitive relation that contains x. When x is a small relation, the transitive closure is hull[TRV, x], but this construction breaks down for proper classes. If x is a proper class, then

$$\mathtt{trv}[\mathtt{x}] = \bigcup \mathtt{image}[\mathtt{HULL}[\mathtt{TRV}], \mathtt{P}[\mathtt{x}]] = \mathtt{image}[\mathtt{power}[\mathtt{x}], \{0\}']$$

is the transitive closure of $\mathtt{x} \cap (\mathtt{V} \times \mathtt{V})$. Here power[x] is a relation whose vertical sections at the natural numbers are the various powers of x,

$$\mathtt{power}[\mathtt{x}] = \mathtt{iterate}[\mathtt{Id} \otimes \mathtt{x}, \mathtt{Id}].$$

Here cross denotes the parallel or cross product of two relations (Belinfante, [1999a]).

The situation here is actually quite similar to the theory of transitive closures of classes tc[x] considered earlier, and indeed there are some connections between the two meanings of transitive. In particular, the transitive closure of the membership relation E is

$$\mathtt{trv}[\mathtt{E}] = \mathtt{inverse}[\mathtt{TC}] \circ \mathtt{E}.$$

Iteration can be used to show that HULL[invar[x]] is a total function when x is thin. When x is thin and y is a set, the relation iterate[x,y] is a set, and hence

$$\mathtt{range}[\mathtt{iterate}[\mathtt{x}, \mathtt{y}]] = \mathtt{y} \cup \mathtt{image}[\mathtt{trv}[\mathtt{x}], \mathtt{y}]$$

is a set which contains y and is invariant under x. In other words, every set is a subset of a set that is invariant under a given thin relation. This fact can be written as follows,

$$\mathtt{thin}[\mathtt{x}] \Rightarrow \mathtt{image}[\mathtt{inverse}[\mathtt{S}], \mathtt{invar}[\mathtt{x}]] = \mathtt{V}.$$

Since the domain of HULL[x] is image[inverse[S], x], this completes the proof that domain[HULL[invar[x]]] = V whenever x is thin. In particular, for the case that x = inverse[E] one deduces that TC = HULL[FULL] is a total function.

The derivations of many properties of the constructor trv require an application of iteration. In particular, this was used to derive the principle of well-founded induction. A relation x is *well-founded* if the only set y satisfying $y \subset \mathtt{image}[x, y]$ is the empty set. For example, the membership relation E is well-founded if and only if the axiom of regularity holds. Whether or not the axiom of regularity holds, the restriction $\mathtt{id}[\Omega] \circ \mathtt{E}$ of the membership relation to the class Ω of ordinals is well-founded. Another familiar example of a well-founded relation is the restriction $\mathtt{id}[\mathtt{FINITE}] \circ \mathtt{PS}$ of the proper subset relation $\mathtt{PS} = \mathtt{S} \cap \mathtt{Id}'$ to the class of finite sets. Since any subclass of a well-founded relation is well-founded, one can show that x is well-founded if and only if $\mathtt{P}[x] \subset \mathtt{WF}$, where WF is the class of all small well-founded relations. The principle of well-founded induction says that if x is a well-founded relation whose inverse is thin, then there are no proper classes y that satisfy $y \subset \mathtt{image}[x, y]$. As an application of this, one can show that if x is a well-founded relation with a thin inverse, then trv[x] is also well-founded. In particular, since any set is thin, it follows that WF is invariant under HULL[TRV].

One of the theorems proved using Otter is the principle of FINITE induction: if the empty set belongs to a class of sets, and if that class is invariant under the cover relation

$$\mathtt{K} = \mathtt{PS} \cap (\mathtt{PS} \circ \mathtt{PS})' = \{\langle x, y\rangle \mid \exists z\ (z \notin x\ \ \&\ \ y = x \cup \{z\})\},$$

then $\mathtt{FINITE} \subset x$. Explicitly:

$$0 \in x\ \ \&\ \ \mathtt{image}[\mathtt{K}, x] \subset x\ \Rightarrow\ \mathtt{FINITE} \subset x.$$

In the course of rederiving this result using the GOEDEL program, the following general formula for the transitive closure of K was also derived:

$$\mathtt{Id} \cup \mathtt{trv}[\mathtt{K}] = \mathtt{CUP} \circ \mathtt{id}[\mathtt{V} \times \mathtt{FINITE}] \circ \mathtt{inverse}[\mathtt{FIRST}].$$

Another simple result along these lines is the formula

$$\mathtt{iterate}[\mathtt{K}, \{0\}] = \mathtt{inverse}[\mathtt{CARD}] \circ \mathtt{id}[\omega],$$

for the relation whose vertical sections at the natural numbers are the classes of all sets with a given cardinality. Here CARD is the cardinality function which assigns to each set the smallest ordinal with which it can be put in one-to-one correspondence, if one exists. Since the axiom of choice is not assumed, the function CARD need not be total, but its domain does contain the class FINITE.

9 Application to Integer Arithmetic

Because the definitions of CORE and HULL functions are not limited to topology, these concepts find important applications in other branches of mathematics. In this section an application to integer addition will be described.

The set Z of all integers can be defined as the set of equivalence classes of a certain equivalence relation EQUIDIFF on the set $\omega \times \omega$ of pairs of natural numbers, where pairs

$\langle \mathtt{u}, \mathtt{v} \rangle$ and $\langle \mathtt{x}, \mathtt{y} \rangle$ are considered to be equivalent if the sum of the natural numbers `u` and `y` equals the sum of `v` and `x`. These equivalence classes are in fact one-to-one functions. For example, the integer zero is the identity function $\mathtt{id}[\omega]$ on the natural numbers, and the integer unity is the successor function on the natural numbers. The non-negative integer `plus[x]` corresponding to the natural number `x` is the function that increments natural numbers by `x`. The negative of the integer `plus[x]` is the function `inverse[plus[x]]`. Each positive integer has domain `omega`, but the domain of `inverse[plus[x]]` is the relative complement of the natural number `x` in ω, that is, the set of all natural numbers greater than or equal to `x`.

The sum of two integers can be defined as the unique integer that contains their composite. As a matter of fact, the composite of two integers is already an integer except for the case that the left factor is positive and the right factor is negative. In that case, the composite is contained in the unique integer obtained by reversing the order of the factors:

$$\mathtt{plus}[\mathtt{x}] \circ \mathtt{inverse}[\mathtt{plus}[\mathtt{y}]] \subset \mathtt{inverse}[\mathtt{plus}[\mathtt{y}]] \circ \mathtt{plus}[\mathtt{x}].$$

This yields the following simple formula for the binary function `INTADD` for integer addition:

$$\mathtt{INTADD} = \mathtt{HULL}[\mathtt{Z}] \circ \mathtt{COMPOSE} \circ \mathtt{id}[\mathtt{Z} \times \mathtt{Z}].$$

From this formula one can derive the familiar properties of integer addition, including the commutative law,

$$\mathtt{INTADD} \circ \mathtt{SWAP} = \mathtt{INTADD},$$

the associative law,

$$\mathtt{INTADD} \circ (\mathtt{Id} \otimes \mathtt{INTADD}) \circ \mathtt{ASSOC} = \mathtt{INTADD} \circ (\mathtt{INTADD} \otimes \mathtt{Id}).$$

The function that takes an integer to its negative is

$$\mathtt{id}[\mathtt{Z}] \circ \mathtt{INVERSE} = \mathtt{INVERSE} \circ \mathtt{id}[\mathtt{Z}].$$

This is an automorphism of integer addition:

$$\mathtt{INVERSE} \circ \mathtt{INTADD} = \mathtt{INTADD} \circ (\mathtt{INVERSE} \otimes \mathtt{INVERSE}).$$

Integer subtraction is expressible in terms of addition and negatives:

$$\mathtt{rotate}[\mathtt{INTADD}] = \mathtt{INTADD} \circ (\mathtt{Id} \otimes \mathtt{INVERSE}).$$

10 Summary

The `CORE` and `HULL` constructors discussed in this paper are useful not only in topology, but have applications in many other branches of mathematics. In group theory, for example, the subgroup generated by a subset of a group is the intersection of all subgroups that contain the given set, a fairly typical application of the hull operation in abstract algebra.

By making available such standard constructors in systems for automated reasoning, and deriving their general properties, a valuable arsenal is created that can be relied upon to furnish the ammunition needed to attack many interesting applications. The lofty dream that automated reasoning and verification systems will one day be used routinely in mathematical reasearch will only be realized if serious efforts are made to connect the abstract principles of automated reasoning with the needs encountered in the everyday practice of modern mathematics, laying a solid foundation upon which one can build the many marvelous edifices that comprise the infrastructure of mathematical research.

Bibliography

[1997] Bancerek, G., Closure Operators and Subalgebras, Journal of Formalized Mathematics, vol. 9 (1997), pp. 295–301.

[1999a] Belinfante, J. G. F., Computer proofs in Gödel's class theory with equational definitions for composite and cross, Journal of Automated Reasoning, vol. 22 (1999) pp. 311–339.

[1999b] Belinfante, J. G. F., On computer-assisted proofs in ordinal number theory, Journal of Automated Reasoning, vol. 22 (1999), pp. 341–378.

[2001a] Belinfante, J. G. F., Computer Proofs about Transitive Closure, in International Joint Conference on Automated Reasoning, IJCAR-2001 Short Papers, pp. 11–20, edited by R. Goré, A. Leitsch and T. Nipkow, Technical Report DII 11/01, Siena, Italy, 19–23 June 2001.

[2001b] Belinfante, J. G. F., Discovering Theorems using GOEDEL: A Case Study, in Calculemus-2001, 9th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 21–22 June 2001, Siena, Italy, edited by Steve Linton and Roberto Sebastiani.

[2003] Belinfante, J. G. F., Reasoning about iteration in Gödel's class theory, in Automated Deduction–CADE-19, Proceedings of the 19th International Conference on Automated Deduction, Miami Beach, FL, USA July–August 2003, edited by F. Baader, Lecture Notes in Artificial Intelligence, vol. 2741, pp. 228–242, Springer Verlag, Berlin, 2003. (ISBN 3-540-40559-3)

[1986] Boyer, R., Lusk, E., McCune, W., Overbeek, R., Stickel M. and Wos, L., Set theory in first orderlogic: clauses for Gödel's axioms, Journal of Automated Reasoning, volume 2 (1986), pages 287–327.

[2004] Chang, K., In Math, Computers Don't Lie. Or Do They?, The New York Times, April 6, 2004.

[1991] Farmer, W. F., and Thayer, F. J., Two Computer-Supported Proofs in Metric-Space Topology, Notices of the American Mathematical Society, vol. 38 (1991), pp. 1133–1138.

[1940] Gödel, K., The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis with the Axioms of Set Theory, Princeton University Press, Princeton, 1940.

[1989] McCune, W. and Wick, C., Automated reasoning about elementary point-set topology, Journal of Automated Reasoning, vol. 5 (1989), pp. 239–255.

[1987] Mendelson, E., Introduction to Mathematical Logic, Third edition, Wadsorth & Brooks/Cole, Monterey, CA, 1987.

[1967] Rubin, J. E., Set Theory for the Mathematician, Holden-Day, San Francisco, 1967.

[1987] Tarski, A., and Givant, S., A Formalization of Set Theory without Variables, American Mathematical Society Colloquium Publications, volume 41, Providence, Rhode Island, 1987.

Classification of Quasigroups by Random Walk on Torus

Smile Markovski, Danilo Gligoroski and Jasen Markovski
(University Ss Cyril and Methodius in Skopje, Macedonia)
{smile,danilo,jasen}@ii.edu.mk

Abstract

Quasigroups are algebraic structures closely related to Latin squares which have many different applications. There are several classifications of quasigroups based on their algebraic properties. In this paper we propose another classification based on the properties of strings obtained by specific quasigroup transformations. More precisely, in our research we identified some quasigroup transformations which can be applied to arbitrary strings to produce pseudo random sequences. We performed tests for randomness of the obtained pseudo-random sequences by random walks on torus. The randomness tests provided an empirical classification of quasigroups.

Key words: random walk, quasigroup transformation, χ^2*-test*

AMS Mathematics Subject Classification (2000): 20N05, 11K45, 62P99

1 Introduction

The classification of finite quasigroups is a problem of big importance considering the applications of quasigroups in many theories like cryptography, coding theory, design theory and others. Two main classifications are obtained by using the algebraic properties of the quasigroups: (1) classes of isotopic quasigroups, which are known only for quasigroups of orders up to 10 [17] and (2) classes of isomorphic quasigroups. Also, quasigroups are classified on varieties according to identities they satisfy (for example, totally symmetric quasigroups, Stein quasigroups, Moufang quasigroups etc.). An interesting algebraic classification of abelian quasigroups is given in [18]. As noted in [16], the classification of algebraic structures like quasigroups is a very important and difficult problem.

In this paper we give a new classification of finite quasigroups based upon the strings obtained by quasigroup transformations, defined in section 2. Several applications of a quasigroup transformation on a given string produce a string which can be considered as pseudo-random sequence. Thus, quasigroup operations on an alphabet may be considered as pseudo-random sequence generators (PRSG). A PRSG designed by using quasigroup transformations (QPRSG) is strongly dependent on the quasigroup operation used in its construction. For some quasigroups the QPRSG generates a random sequence that passes all publicly available tests for pseudo-random sequences, but there are quasigroups that produce sequences far away from random ones. Our classification is based on statistical test for randomness defined by using random walks on torus. Given an alphabet $A = \{a_1, a_2, \dots, a_n\}$ and a quasigroup operation on A we transform the string $a_1a_2\dots a_na_1a_2\dots a_n\dots a_1a_2\dots a_n$ into a possible pseudo random sequence. Afterwards we measure how the newly obtained sequence performs on the tests, which provides an empirical classification of finite quasigroups of any order. The organization of the paper is as follows. Needed facts on quasigroups and quasigroup string transformations are given in section 2. The random walk on torus and the corresponding statistical tests are considered in section 3. Section 4 describes the program support we have used. It presents an original software design for the purposes of this research that can be freely downloaded from the following URL *http://twins.ii.edu.mk/trw*). A complete classification of quasigroups of order 4 is presented in section 5, and in section 6 we present some results obtained for classification of quasigroups of higher order. There are 3 appendices as well. Periodicity of the quasigroup transformed strings and QPRSG are discussed in appendix 1. In appendix 2 we present graphical representations of the random walks on torus for some quasigroups.

2 Quasigroup string transformations

A quasigroup is a groupoid $(Q,*)$ satisfying the law

$$(\forall u,v\in Q)(\exists! x,y\in Q)(u*x=v\ \&\ y*u=v).$$

This implies the cancellation laws $x*y=x*z \implies y=z,\ y*x=z*x \implies y=z$ and the equations $a*x=b,\ y*a=b$ have unique solutions $x,\ y$ for each $a,b\in Q$.

Given a quasigroup $(Q,*)$ five so called parastrophes (or conjugate operations) can be adjoint to $*$, and here we will use only two of them, denoted by $\backslash$ and $/$ and defined by

$$x*y=z \iff y=x\setminus z \iff x=z/y \tag{2}$$

Then $(Q,\backslash)$ and $(Q,/)$ are quasigroups too and the algebra $(Q,*,\backslash,/)$ satisfies the identities

$$x\setminus(x*y)=y,\quad (x*y)/y=x,\quad x*(x\setminus y)=y,\quad (x/y)*y=x \tag{3}$$

Conversely, if an algebra $(Q,*,\backslash,/)$ with three binary operations satisfies the identities (3), then $(Q,*)$, $(Q,\backslash)$, $(Q,/)$ are quasigroups and (2) holds.

A Latin square on a finite set Q of cardinality $|Q|=n$ is an $n\times n$-matrix with entries from Q such that each row and each column of the matrix is a permutation of Q. To any finite quasigroup $(Q,*)$ given by its multiplication table a Latin square can be associated, consisting of the matrix formed by the main body of the table, since each row and column of the matrix is a permutation of Q.

Using quasigroups several quasigroup string transformations can be defined and here we will define only two of them. Consider an alphabet (i.e. a finite set) A, and denote by A^+ the set of all nonempty words (i.e. finite strings) formed by the elements of A. The elements of A^+ will be denoted by $a_1a_2\dots a_n$ rather than $(a_1,a_2,\dots,a_n)$, where $a_i\in A$. Let $*$ be a quasigroup operation on the set A, i.e. consider a quasigroup $(A,*)$. For each $l\in A$ we define functions $e_l,\ e'_l : A^+\longrightarrow A^+$ as follows. Let $a_i\in A,\ \alpha=a_1a_2\dots a_n$. Then

$$e_l(\alpha)=b_1\dots b_n \iff b_{i+1}=b_i*a_{i+1},$$

$$e'_l(\alpha) = b_1 \dots b_n \Longleftrightarrow b_{i+1} = a_{i+1} * b_i$$

for each $i = 0, 1, \dots, n-1$, where $b_0 = l$.

The functions e_l, e'_l are called e- and e'-transformation of A^+ based on the operation $*$ with leader l.

The compositions of mappings

$$E_k = e_{l_1} \circ e_{l_2} \circ \dots \circ e_{l_k},$$

and

$$E'_k = e'_{l_1} \circ e'_{l_2} \circ \dots \circ e'_{l_k},$$

where l_i are leaders, are said to be E- and E'-transformations of A^+ respectively. Further on we will usually use only one leader, i.e. $l = l_i$ for each i.

Example 1 Let $A = \{a, b, c, d\}$ and let the quasigroup operation $*$ on A be defined by

$*$	a	b	c	d
a	c	b	d	a
b	a	c	b	d
c	d	a	c	b
d	b	d	a	c

Take a to be the leader and $\alpha = bbbbaccdaadbdcabdbdcaaa$. Then the transformed strings $E_1(\alpha) = e_a(\alpha)$, $E_2(\alpha) = e_a(E_1(\alpha)), E_3(\alpha) = e_a(E_2(\alpha))$ are the following:

	$b\ b\ b\ b\ a\ c\ c\ d\ a\ a\ d\ b\ d\ c\ a\ b\ d\ b\ d\ c\ a\ a\ a$	$= \alpha$
a	$b\ c\ a\ b\ a\ d\ a\ a\ c\ d\ c\ a\ a\ d\ b\ c\ b\ c\ b\ b\ a\ c\ d$	$= E_1(\alpha)$
a	$b\ b\ a\ b\ a\ a\ c\ d\ a\ a\ d\ b\ a\ a\ b\ b\ c\ c\ a\ b\ a\ d\ c$	$= E_2(\alpha)$
a	$b\ c\ d\ d\ b\ a\ d\ c\ d\ b\ d\ d\ b\ a\ b\ c\ c\ c\ d\ d\ b\ d\ a$	$= E_3(\alpha)$

□

The functions E_k and E'_k have the following properties [9, 10]:

Theorem 1 *The transformations E_k and E'_k are permutations of A^+.* □

Theorem 2 *Consider an arbitrary string $\alpha = a_1 a_2 \dots a_n \in A^+$, where $a_i \in A$, and let $\beta = E_k(\alpha)$, $\beta' = E'_k(\alpha)$. If n is sufficiently large integer then, for each $s :\ 1 \leq s \leq k$, the distribution of substrings of β and β' of length s is uniform. (We note that for $s > k$ the distribution of substrings of β and β' of length s may not be uniform.)* □

We say that a string $\alpha = a_1 a_2 \dots a_n \in A^+$, where $a_i \in A$, has a period p if p is the smallest positive integer such that $a_{i+1}a_{i+2}\dots a_{i+p} = a_{i+p+1}a_{i+p+2}\dots \ \dots a_{i+2p}$ for each $i \geq 0$.

Let α, β, β' be as in Theorem 1. In appendix 1 we prove the following theorem:

Theorem 3 *The periods of the strings β and β' are increasing at least linearly by k.* □

The increase of the periods depends of the quasigroup operations, and as seen from our experiments (and also by [2]) for some quasigroups $(Q, *)$ it is exponential, i.e. if α has a period p, then $\beta = E_k(\alpha)$ and $\beta' = E'_k(\alpha)$ may have periods greater than pq^k for some real number $q: \ |Q| \geq q > 1$. In such a way the class of finite quasigroups can be separated into two subclasses: the class of quasigroups with exponential growth (exponential quasigroups) and the class of quasigroups with linear growing (linear quasigroups). There are no known criteria for distinguishing these two classes of quasigroups. By many experiments we have made it can be noticed that only the exponential quasigroups produce good pseudo-random sequences.

In what follows we will usually use only E-transformations, since the results will hold for E'-transformations by symmetry.

3 Statistical tests of randomness by using random walk on torus

Random walks are defined on the discrete plane $\mathbb{Z}^2$. Given a (pseudo) random sequence, a random walk can be defined in many different ways. If the random sequence has elements from the alphabet $\{a, b, c, d\}$ then we can use the four one-step directions left (when a appears), right (when b appears), up (for c) and down (for d). For the alphabet $\{a, b, c, d, e\}$ we can choose the stop option (no movement) if e appears, and for 8 letter alphabet we can choose the diagonal movements as well. For 6 letter alphabet $\{a, b, c, d, e, f\}$ we can choose a diagonal movement up and left when e appears, and a diagonal movement up and right when f appears. In the case of 7 letter alphabet $\{a, b, c, d, e, f, g\}$ we can add stop option for g. For an alphabet with more than 8 letters we can group the letters in classes each one containing 4, 5, 6, 7 or 8 letters, and then we can use the preceding definitions of movements. If we have two or three letter alphabet we will consider pairs of letters as one letter.

One can argue that the movements defined for 6 and 7 letter alphabet require very large discrete planes (demanding huge memory arrays), which makes them difficult for designing suitable program support. That is why we have chosen a random walk on a torus. Instead of the whole discrete plane we take the square bordered by the points with coordinates $(-n, -n)$, $(-n, n)$, (n, n) and $(-n, n)$, where n is a positive integer, and we identify the points (s, n) and $(s, -n)$ for each $s: \ -n \leq s \leq n$, and the points (n, t) and $(-n, t)$ for each $t: \ -n \leq t \leq n$. Then we say that the torus size is n.

The random walks can be used for designing many suitable tests for PRSGs (see for example [11, 19]). We suppose that each point (x, y) of the discrete torus has a weight 0 at the beginning, and we increase the weights of the points according to the definitions of the movements, following the next procedure. Let s be a fixed positive integer. For a given sequence $\alpha = a_1a_2 \dots a_d$, starting from the coordinate center $(0, 0)$ we make s steps according to the values of the first s elements $a_1a_2 \dots a_s$ and we add 1 to the weight of the point (p, q) where the movement stopped. After that, starting from the point (p, q), we continue the movement following the next s elements $a_{s+1} \dots a_{2s}$ of the string α and we increase the weight of the point (u, v) where the movements stopped, then we continue starting from the point (u, v), and so on. Note that the average weight of a point, i.e. the frequency of stops at that point, is $f = d/s$ and we choose d and s

such that f is an integer.

For a given pseudo-random sequence, we can count the weights of the points of the torus. On the other hand, assuming that we have a theoretically perfect random sequence, we can count the weights (i.e. the frequency) as a product of the probability of the stop at the point (p, q) and the number of trials, obtaining in such a way the theoretical frequency of stops. Further on we will use the following property [15]:

Proposition 1 *The distribution of the weights obtained from unbiased random sequence is uniform.* □

The statistical tests are defined as follows. We divide a torus of size n on t regions with equal number of discrete points by using r parallel horizontal and k parallel vertical lines (rows and columns), where r and k are factors of n. By Proposition 1 the theoretical weights of each region is $E_i = fn^2/t$ for $i = 1, \dots, t$. We compare the random sequences obtained by PRNGs with the theoretical ones by using the Pearson χ^2-test, where the test statistics is given by $\chi^2 = \sum_{i=0}^{t-1} \frac{(O_i - E_i)^2}{E_i}$, and it has χ^2 distribution with $t-1$ degrees of freedom, where O_i denotes the number of arrivals at i-th region from a sequence obtained by a PRNG. We accept the assumption that the random sequence generated by PRNG is uniformly distributed if $\chi^2 \leq \chi^2_{t-1,p}$, where $\chi^2_{t-1,p}$ is a number which satisfy the condition $P\{\chi^2 > \chi^2_{t-1,p}\} = p$, for given p. In opposite case, we reject the assumption of uniformity. Note that the statistics will be relevant only if we have sufficiently large sequences.

4 Program support

We performed the experiments on the quasigroups using a Java application that (1) generates quasigroups, (2) performs a random walk using the generated quasigroups, (3) does the required statistics on the obtained random walk data and (4) generates visual representation of the obtained random walk data in bitmap format (.BMP). The software package requires standard Java run-time environment (JRE) and it can be freely downloaded from the following URL *http://twins.ii.edu.mk/trw*).

The user input is defined by text configuration files. There are three configuration files, one for each type of operation: (1) quasigroup generation, (2) random walk and (3) statistical tests. The application reads the configuration files and performs the operations defined in them. For example, the configuration file for the quasigroup generation contains two main fields: (1) *QGOrder*, the order of the quasigroup and (2) *GenerationMethod*, which can be A - all quasigroups of a given order, L - predefined portion of quasigroups in lexicographic order (e.g. from 50th until 60th quasigroup) and R - predefined number of randomly generated quasigroups. In case of lexicographic generation the user has to enter the starting number and the ending number of the quasigroup (using the fields *LexiStart* and *LexiEnd*) and in case of random generation the user has to enter the number of randomly generated quasigroups (using the field *RandomGeneration*).

The quasigroups are exported in a text file which allows easy manipulation. The names of the files reveals the order of the quasigroups and the way the quasigroups have been generated. For example, QG_o8_R_6.txt contains 6 random quasigroups of order 8. For more details, please refer to the readme file available in the software package.

The random walk data is also exported in a text file with the according quasigroup used to perform the random walk. Again, the text file is named with a special format. The first part of the filename is the same as the file that contains the quasigroups. Afterwards, there is information about the size of the torus and the number of expected arrivals per torus point. For example, if we perform a torus walk on the previous example for a torus with size 100×100 points and we expect 200 visits per point, the random walk output filename will be QG_o8_R_6_w100_e200.txt. For more details, please refer to the readme file available in the software package.

The random walks require user input values for the following parameters: (*i*) torus size, (*ii*) random walk length, (*iii*) random walk definition (left, right, up, down, stop,...), (*iv*) number of expected visits, (*v*) number of applications of the transformation E and (*(vi)*) the leader. Since we wanted to test different types of walks, the walk is defined using relative coordinate changes in regard to the number of the pseudo random sequence.

For example, if the letter 2 means the walk continues in the direction up, than we define the relative coordinate change as (0, 1) (down would be (0, -1), left (-1, 0), stop (0, 0), left-up (-1,1) etc). In this way the user is able to define any type of random walk.

An additional feature allows generation of bitmap images that give a visual presentation of the random walk. The bitmaps are generated automatically using the random walk data. Each pixel presents one point on the torus body. Brighter pixels denote points with higher number of arrivals. The brightness of the pixels is calculated relatively to the minimal and maximal number of arrivals on the whole torus body.

The application assumes an unified alphabet for each quasigroup of given order n to be $\{0,\ 1\ \ldots,\ n-1\}$. It performs the randomness test of the pseudo random number sequence using the χ^2 fit test as described previously. More precisely it calculates the sum as defined for χ^2 statistics and exports the obtained results in a log text file which contains information about the quasigroup and the obtained and expected χ^2 values.

The χ^2 statistics can be performed using different partitioning of the torus body on regions and different probabilities of success. The program contains predefined χ^2 values for degrees of freedom from 1 to 30 and from 40 to 100 with step 10 and for probabilities to fit: 0.1, 0.075, 0.05, 0.01, 0.005, which can be found in any statistical textbook. Furthermore, the user can enter the χ^2 values which are not embedded in the program. Thus, there is no limit on the number of statistics which can be performed on a given random walk.

The number of expected visits per point defines the expected stops in case of perfect random number sequence. The number of applications of the transformation E defines how many transformations have to be applied in order to obtain the pseudo-random sequence. These two parameters together with the torus size and the random walk length directly influence the number of operations required to perform the random walk. Higher numbers give more accurate tests, but require much more time. The complexity of the random walk is given by the following equation

$$\begin{aligned} &Total\ number\ of\ operations = \\ &\qquad = Torus\ points * Walk\ length * \\ &\qquad * Expected\ stops * Number\ of\ E\ transformations \end{aligned}$$

Thus, the execution time of the program is polynomial, but it should not be underestimated since the calculation of the random walk data took about two days for all quasigroups of order 4, for (1) torus with 40×40 points, (2) 300 expected visits per point, (3) 121 steps of the random walk and (4) 50 E-transformations, on a Pentium 4 server with 2.8GHz and 1 GB RAM.

5 Classification of quasigroups of order 4

There are 576 quasigroups of order 4 and they are lexicographically ordered. The lexicographic ordering of the quasigroups of a given order is performed in such a way that the rows of the Latin square are concatenate the-next-after-the-previous and then lexicographic ordering of that sequences is applied. The i-th quasigroup of order n in the lexicographic ordering is denoted by QGn-i. The first seven quasigroups of order 4 are the following ones

0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
1 0 3 2	1 0 3 2	1 0 3 2	1 0 3 2	1 2 3 0	1 2 3 0	1 3 0 2
2 3 0 1	2 3 1 0	3 2 0 1	3 2 1 0	2 3 0 1	3 0 1 2	2 0 3 1
3 2 1 0	3 2 0 1	2 3 1 0	2 3 0 1	3 0 1 2	2 3 0 1	3 2 0 1
QG4-1	QG4-2	QG4-3	QG4-4	QG4-5	QG4-6	QG4-7

where the main row and column are 0 1 2 3.

We have used 8 statistical test based on random walk of torus defined as follows. The torus size was taken $n = 40$, and that means that 1600 discrete points had to be visited by the random walks, with an average of 300 stops at each one, after 121 movements left, right, up or down. The E-transformations were taken 50 times over the string 01230123...0123 with leader 0. We divided the torus on 4 types of regions R_1, R_2, R_3, R_4 with equal number of points, where R_1 consists of 4 regions, R_2 of 8 regions, R_3 of 16 regions and R_4 of 25 regions. We took for each type of region two statistics with following values for p and χ^2:

R_1 − $S1 : p = 0.050,\ \chi^2 = 9.488$ with 3 degree of freedom,
R_1 − $S2 : p = 0.075,\ \chi^2 = 8.496$ with 3 degree of freedom,
R_2 − $S3 : p = 0.050,\ \chi^2 = 15.507$ with 7 degree of freedom,
R_2 − $S4 : p = 0.075,\ \chi^2 = 14.270$ with 7 degree of freedom,
R_3 − $S5 : p = 0.050,\ \chi^2 = 26.296$ with 15 degree of freedom,
R_3 − $S6 : p = 0.100,\ \chi^2 = 23.543$ with 15 degree of freedom,
R_4 − $S7 : p = 0.050,\ \chi^2 = 37.653$ with 24 degree of freedom,
R_4 − $S6 : p = 0.100,\ \chi^2 = 34.382$ with 24 degree of freedom,

From the obtained results we concluded that 200 quasigroups failed all tests and they are linear quasigroups, while 376 passed the tests and they are exponential quasigroups. We have applied 50 E-transformations in order to obtain more accurate classification of the quasigroups.

We classified the exponential quasigroups according to the numbers of tests failed, so we obtained 8 classes. Here we also put the quasigroups that failed on all 8 tests, since they are exponential ones. Namely, they passed all the tests when 200 E-transformations were used instead of 50.

Quasigroups that failed on 0 **tests**: QG4-6, 8, 17, 20, 22, 23, 29, 33, 35, 36, 38, 50, 56, 59, 61, 62, 69, 73, 74, 75, 78, 79, 81, 84, 86, 87, 88, 90, 91, 94, 96, 99, 103, 105, 107, 117, 119, 120, 123, 125, 134, 135, 136, 140, 141, 143, 150, 151, 155, 156, 158, 165, 175, 177, 181, 184, 186, 188, 190, 199, 200, 204, 205, 210, 215, 216, 217, 224, 225, 227, 231, 236, 241, 245, 247, 248, 250, 254, 260, 264, 270, 271, 276, 277, 278, 279, 280, 282, 283, 288, 289, 290, 298, 304, 309, 311, 317, 321, 326, 327, 336, 337, 338, 347, 350, 352, 357, 358, 360, 362, 367, 368, 369, 372, 373, 383, 384, 387, 390, 391, 393, 396, 400, 404, 410, 415, 416, 418, 419, 422, 424, 426, 427, 428, 434, 442, 443, 446, 448, 449, 453, 458, 468, 469, 470, 473, 474, 475, 478, 479, 481, 482, 486, 492, 493, 502, 509, 515, 522, 525, 529, 539, 542, 543, 547, 555, 557, 558, 562, 564, 565, 567, 571

Quasigroups that failed on 1 **test**: QG4-12, 31, 41, 85, 104, 106, 114, 115, 153, 161, 167, 180, 183, 193, 198, 207, 209, 230, 233, 240, 257, 258, 268, 281, 297, 299, 300, 301, 310, 316, 320, 333, 339, 351, 356, 361, 366, 386, 412, 413, 421, 425, 441, 447, 454, 505, 511, 513, 548, 554

Quasigroups that failed on 2 **tests**: QG4-10, 13, 44, 53, 64, 72, 76, 95, 108, 128, 129, 149, 194, 195, 202, 238, 249, 266, 286, 287, 307, 319, 328, 329, 330, 340, 437, 455, 460, 465, 471, 783, 503, 516, 527, 533

Quasigroups that failed on 3 **tests**: QG4-15, 30, 34, 52, 65, 159, 191, 323, 344, 353, 382, 423, 435, 457, 489, 490, 504, 510, 518, 538

Quasigroups that failed on 4 **tests**: QG4-19, 45, 47, 112, 154, 162, 173, 208, 244, 255, 273, 332, 346, 370, 375, 377, 389, 463, 472, 491, 498, 541, 546, 560, 569

Quasigroups that failed on 5 **tests**: QG4-66, 109, 201, 221, 239, 267, 296, 313, 379, 394, 452, 544

Quasigroups that failed on 6 **tests**: QG4-58, 131, 164, 211, 214, 220, 237, 251, 322, 341, 363, 378, 440, 459, 462, 521, 524, 530

Quasigroups that failed on 7 **tests**: QG4-67, 89, 98, 122, 187, 226, 261, 295, 312, 480, 499, 501, 512, 535

Quasigroups that failed on 8 **tests**: QG4-32, 39, 102, 118, 124, 137, 152, 168, 219, 256, 265, 294, 306, 376, 397, 402, 409, 436, 487, 488, 496, 508, 519, 532, 536, 545, 561

The linear quasigroups can be grouped in much more subclasses. For that aim it is not enough to see only how they failed the tests. There are quasigroups with same experimentally obtained χ^2 values and they are in the same class as well. Nevertheless, there are quasigroups with similar experimental results and they were classified in classes by using the visual presentations of the distributions of the stops at the torus. Our program support allows bitmap images of the toruses and similarities of the test results and similarities of the images were used for suitable classification. We obtained 64 different classes, but we have to stress out that these classes are results of our choice of the parameters made with an intention as much as possible classes to be obtained. If we choose for instance a torus with side 100 and 200 E-transformations with leader 3, we could obtained a somewhat different classification. We named the classes as $\mathbf{C}_i$, where QG4-i is the first quasigroup that appears in that class. We start with classes of higher cardinality.

$\mathbf{C}_1 = \{QG4-1, 4, 11, 24, 26, 27, 42, 48, 51, 57, 68, 126, 139, 142\}$
$\mathbf{C}_{246} = \{QG4-246, 318, 408, 520, 526, 566, 576, 430, 438, 550, 431, 551\}$
$\mathbf{C}_{157} = \{QG4-157, 163, 196, 243, 252, 315, 420, 476, 485, 517, 253\}$
$\mathbf{C}_{146} = \{QG4-146, 147, 169, 172, 178, 218, 229, 259, 388, 331\}$
$\mathbf{C}_{14} = \{QG4-14, 21, 92, 113, 203, 285, 305, 364, 385\}$
$\mathbf{C}_{176} = \{QG4-176, 182, 223, 232, 433, 456, 553, 559, 572\}$
$\mathbf{C}_{16} = \{QG4-16, 40, 43, 60, 70, 130, 133, 138\}$
$\mathbf{C}_{174} = \{QG4-174, 263, 335, 392, 514, 528, 568, 570\}$

$\mathbf{C}_{179} = \{QG4 - 179, 192, 334, 374, 464, 494, 653\}$
$\mathbf{C}_{189} = \{QG4 - 189, 348, 354, 395, 405, 573\}$
$\mathbf{C}_{235} = \{QG4 - 235, 342, 364, 467, 495, 507\}$
$\mathbf{C}_{407} = \{QG4 - 407, 429, 432, 549, 552, 575\}$
$\mathbf{C}_{2} = \{QG4 - 2, 3, 18, 25, 28\}$
$\mathbf{C}_{253} = \{QG4 - 253, 272, 292, 381, 500\}$
$\mathbf{C}_{7} = \{QG4 - 7, 9, 49, 63\}$
$\mathbf{C}_{83} = \{QG4 - 83, 111, 398, 556\}$
$\mathbf{C}_{262} = \{QG4 - 262, 325, 411, 534\}$
$\mathbf{C}_{345} = \{QG4 - 345, 359, 399, 401\}$
$\mathbf{C}_{5} = \{QG4 - 5, 121, 144\}$
$\mathbf{C}_{145} = \{QG4 - 145, 170, 171\}$
$\mathbf{C}_{166} = \{QG4 - 166, 349, 439\}$
$\mathbf{C}_{213} = \{QG4 - 213, 303, 324\}$
$\mathbf{C}_{228} = \{QG4 - 228, 444, 537\}$
$\mathbf{C}_{55} = \{QG4 - 55, 71\}$
$\mathbf{C}_{97} = \{QG4 - 97, 116\}$
$\mathbf{C}_{101} = \{QG4 - 101, 302\}$
$\mathbf{C}_{160} = \{QG4 - 160, 206\}$
$\mathbf{C}_{185} = \{QG4 - 185, 403\}$
$\mathbf{C}_{197} = \{QG4 - 197, 380\}$
$\mathbf{C}_{242} = \{QG4 - 242, 314\}$
$\mathbf{C}_{284} = \{QG4 - 284, 540\}$
$\mathbf{C}_{343} = \{QG4 - 343, 484\}$
$\mathbf{C}_{365} = \{QG4 - 365, 497\}$
$\mathbf{C}_{406} = \{QG4 - 406, 574\}$
$\mathbf{C}_{445} = \{QG4 - 445, 506\}$

The rest of the classes consist of only one quasigroup:

$\mathbf{C}_{37},\ \mathbf{C}_{46},\ \mathbf{C}_{54},\ \mathbf{C}_{77},\ \mathbf{C}_{80},\ \mathbf{C}_{82},\ \mathbf{C}_{93},\ \mathbf{C}_{100},\ \mathbf{C}_{110},\ \mathbf{C}_{127},\ \mathbf{C}_{132},\ \mathbf{C}_{212},\ \mathbf{C}_{222},$
$\mathbf{C}_{234},\ \mathbf{C}_{269},\ \mathbf{C}_{274},\ \mathbf{C}_{275},\ \mathbf{C}_{293},\ \mathbf{C}_{308},\ \mathbf{C}_{355},\ \mathbf{C}_{371},\ \mathbf{C}_{414},\ \mathbf{C}_{417},\ \mathbf{C}_{450},\ \mathbf{C}_{451},$
$\mathbf{C}_{461},\ \mathbf{C}_{466},\ \mathbf{C}_{477},\ \mathbf{C}_{531}$

On Appendix 3 one can see graphical representations of some of the classes given by the distributions of the stops over the toruses.

We have to stress out that the preceding classification depends on our choice of the parameters. The main classification of 200 linear and 376 exponential quasigroups will be not changed under any choice of parameters. The only changes can be obtained in classification of linear quasigroups.

6 Results for quasigroups of higher order

We made experiments with the first 500 quasigroups (in lexicographic ordering) of order 5, 6, 7 and 8. The tests were made with same parameters as for quasigroups of order 4.

Quasigroups of order 5: All quasigroups are exponential except QG5-77, QG5-119, QG5-145, QG5-213, QG5-241, QG5-285, and all of these are commutative loops, and QG5-372, QG5-402, QG5-472, that are obtained from commutative loops with permutated rows. Of course, there are quasigroups of order 5 that are linear and that are not loops (with permutated rows).

Quasigroups of order 6: There are 43 quasigroups that did not pass the tests. The experimental results differ from the theoretical in hundreds for quasigroups QG6-2, 6, 36, 48, 53, 65, 79, 100, 101, 107, 171, 195, 217, in thousands for quasigroups QG6-1, 4, 5, 49, 50, 52, 54, 56, 97, 99, 102, 103, 104, 146, 147, 149, 152, 193, and in ten of thousands for quasigroups QG6-3, 6, 7, 8, 51, 55, 98, 145, 148, 150. It is interesting that QG6-157 passed the tests with χ^2 values approximately equal to 0, but still it is a linear one, and that can be seen from its "torus image". (It is a result of an inappropriate choice of the regions.) So, the choice of the parameters and the regions has influence in the classification.

Quasigroups of order 7: All quasigroups of order 7 passed the tests. Still, there are linear quasigroups of order 7.

Quasigroups of order 8: In the first 500 quasigroups of order 8 there are 63 linear, and QG8-1, 2, 25 are commutative loops. QG8-44, 105, 496 failed with hundred, QG8-12, 16, 56, 66, 84, 130, 193, 205, 209, 213, 241, 253, 265, 273, 281, 324, 420, 432 failed with thousand, QG8-3, 4, 6, 7, 8, 9, 10, 11, 13, 14, 15, 17, 29, 31, 33, 41, 53, 81, 93, 140, 145, 157, 161, 165, 284, 293, 2 96, 321, 333, 395, 481, 484 failed with ten of thousand, while QG8-5, 27, 97, 389, 399, 493 failed with million.

One conclusion of the presented results may be that quasigroups of prime order tend to be more exponential than the quasigroups of composite order.

7 Conclusion

We showed that the classification of the quasigroups can be made by using suitable experiments, and that the algebraic way of classifications is not the only useful one. In our classifications, in fact, the algebraic properties of quasigroups do not have exclusive importance. Thus, QG4-1 and QG4-172 are isomorphic ones, but they are classified in two different classes, namely QG4-172 belongs to the class $\mathbf{C}_{146}$. On the other hand, it seems that all commutative loops are linear quasigroups, but that assertions is not proved.

Why it is important to have such classifications? We noted already the importance of exponential quasigroups as tools for QPRSG. They are also important in construction of other cryptographic tools like stream cipher or hash functions [4, 7, 9, 12]. The linear quasigroups are used in design theory, for example in constructions of Steiner triple systems [6]. Also, the symmetries that appear in some linear quasigroups can be used for construction of classes of quasigroups of huge order (2^{1024} for example) that are needed in some applications [3, 8].

Bibliography

[1] Dénes, J., Keedwell, A.D.: Latin Squares and their Applications, English Univer. Press Ltd., 1974

[2] Dimitrova, V., Markovski J.: On quasigroup pseudo random sequence generator,Proc. of the 1-st Balkan Conference in Informatics, Y.Manolopoulos and P. Spirakis eds., 21-23 Nov. 2004, Thessaloniki, pp. 393–401

[3] Gligoroski, D.: Stream cipher based on quasigroup string transformations in $\mathbb{Z}_p^*$, Contributions, Sec. Math. Tech. Sci., MANU (in print)

[4] Gligoroski, D., Markovski, S., Bakeva, V.: On Infinite Class of Strongly Collision Resistant Hash Functions "EDON-F" with Variable Length of Output, Proc. 1-st Inter. Conf. Mathematics and Informatics for industry MII 2003, 14-16 April, Thessaloniki, 302–308

[5] Gligoroski, D., Markovski, S.: Potential of quaisgroups as PRSG (in preparation)

[6] Goračinova-Ilieva, L., Markovski, S. and Sokolova, A.: On groupoids with identity $x(xy) = y$, Quasigroups and Related systems **11** (2004), 39–54

[7] Markovski, S.: Quasigroup string processing and applications in cryptography, Proc. 1-st Inter. Conf. Mathematics and Informatics for industry MII 2003, 14-16 April, Thessaloniki, 278–290

[8] Markovski, S., Gligoroski, D.: Construction of quasigroups of huge order (in preparation)

[9] Markovski, S., Gligoroski, D., Andova, S.: Using quasigroups for one-one secure encoding, Proc. VIII Conf. Logic and Computer Science "LIRA '97", Novi Sad, (1997) 157–162

[10] Markovski, S., Gligoroski, D., and Bakeva, V.: Quasigroup string processing: Part 1, Contributions, Sec. Math. Tech. Sci., MANU, XX 1-2(1999) 13–28.

[11] Markovski, S., Gligoroski, D. and Bakeva, V.: Random walk tests for pseudo-random number generators, Mathem. Commun. **6**(2001) No.2, 135–143

[12] Markovski, S., Gligoroski, D., Stojčevska, B.: Secure two-way on-line communication by using Quasigroup Enciphering with almost public key, Novi Sad J. Math. Vol. 30, No.2, 2000, 43–49

[13] Markovski, S., Kusakatov, V.: Quasigroup string srocessing: Part 2, Contributions, Sec. math. Tech.Sci., MANU, XXI, 1-2(2000) 15–32

[14] Markovski, S., Kusakatov, V.: Quasigroup string processing: Part 3, Contributions, Sec. math. Tech.Sci., MANU, XXII, 1(2001) (in print)

[15] Markovski, S., Mihova, M.: Statistical tests for randomness using random walks on torus (in preparation)

[16] McCasland, R.L., Sorge, V.: Automating Algebra's Tedious Tasks: Computerised Classification, Proc. First Workshop on Challenges and Novel Applications for Automated Reasoning, Miami,2003 (http://www.uclic.ucl.ac.uk/usr/jgow/cnaar.pdf) 37–40

[17] McKay, B.D., Rogoyski, E.: Latin squares of order 10, Electronic J. Comb. **2** (1995) http://ejc.math.gatech.edu:8080/Journal/journalhome.html

[18] Schwenk, J.: A classification of abelian quasigroups, Rendiconti di Matem., Serie VII, V.15, Roma (1995), 161–172

[19] Vattulainen I. and Ala-Nissila, T.: *Mission Impossible: Find a Random Pseudorandom Number Generator*, Computers in Physics, Vol.9, No. 5, 1995, 500-504

[20] http://www.csis.hku.hk/ diehard/

Appendix 1: Periodicity of the quasigroup transformed strings and QPRSG

Let A be a finite alphabet of cardinality n and $*$ be a quasigroup operation on A. Take a fixed element $a \in A$ such that $a*a \neq a$ as a leader, and consider the string $\alpha = a_1 \dots a_k$ where $a_i = a$ for each $i \geq 1$ and k is sufficiently large. Hence, the period of the string α is 1. Apply the transformation E_s on α and denote $E_s(\alpha) = a_1^{(s)} \dots a_k^{(s)}$. The results are presented on Table 1.

	a	a	$\dots$	a	a	$\dots$
a	a'_1	a'_2	$\dots$	a'_{p-1}	a'_p	$\dots$
a	a''_1	a''_2	$\dots$	a''_{p-1}	a''_p	$\dots$
a	a'''_1	a'''_2	$\dots$	a'''_{p-1}	a'''_p	$\dots$
a	$a_1^{(4)}$	$a_2^{(4)}$	$\dots$	$a_{p-1}^{(4)}$	$a_p^{(4)}$	$\dots$
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	

Table 1

We have that $a'_p = a$ for some $p > 1$ since $a*a \neq a$ and $a'_i \in A$ (so we have that p is at least n), and let p be the smallest integer with this property. It follows that the string $E_1(\alpha)$ has a period p. For similar reasons we have that each of the strings $E_s(\alpha)$ is periodical. We will show that it is not possible all of the strings $E_s(\alpha)$ to be of same period p (except in the case when a generates a subquasigroup of order 2.) If we suppose that it is true, we will have $a_p^{(s)} = a$ for each $s \geq 1$. Then we will also have that there are $b_i \in A$ such that the following equalities hold:

$$\begin{array}{ll} a_{p-1}^{(s)} = b_{p-1} & \text{for } s \geq 2 \\ a_{p-2}^{(s)} = b_{p-2} & \text{for } s \geq 3 \\ \vdots & \\ a_1^{(s)} = b_1 & \text{for } s \geq p \end{array}$$

Then we have that $a * b_1 = b_1$, and that implies $a_1^{(s)} = b_1$ for each $s \geq 1$. We obtained $a*a = a*b_1 = b_1$, implying $a = b_1$, a contradiction with $a*a \neq a$. As a consequence we have that $a_1^{(p+1)} = a * a_1^{(p)} = a * b_1 \neq b_1$, $a_2^{(p+1)} = a_1^{(p+1)} * b_2 \neq b_2$, $\dots, a_{p-1}^{(p+1)} = a_{p-2}^{(p+1)} * b_{p-1} \neq b_{p-1}$, $a_p^{(p+1)} = a_{p-1}^{(p+1)} * a \neq a$. We conclude that the period of the string $E_{p+1}(\alpha)$ is not p.

We will show that if a string $\beta \in A^+$ has a period p and $\gamma = E(\beta)$ has a period q, then p is a factor of q. Namely, if $\gamma = b_1 \dots b_q b_1 \dots b_q \dots b_1 \dots b_q$, then $\beta = (a \setminus b_1)(b_1 \setminus b_2) \dots (b_{q-1} \setminus b_q)||(b_q \setminus b_1)(b_1 \setminus b_2) \dots (b_{q-1} \setminus b_q)|| \dots ||(b_q \setminus b_1)(b_1 \setminus b_2) \dots (b_{q-1} \setminus b_q)$ is a periodical string with period $\leq q$. So, $p \leq q$ and this implies that p is a factor of q.

Combining the preceding results, we proved the following theorem for a string α with period p_0:

Theorem 4 *The strings $E_s(\alpha)$ are periodical with periods p_s that are multiples of p_0. The periods p_s satisfy the inequality*

$$p_{p_{s-1}} > p_{s-1} \tag{4}$$

for each $s \geq 1$. □

We note that instead of the transformation E one can consider the transformation E'. Then, the table obtained for E' can be obtained from the Table 1 if it is transformed symmetrically by the main diagonal. This implies that the following theorem is also true:

Theorem 5 *The strings $\alpha_i = a'_i a''_i \dots a_i^{(s)} \dots$ are periodical with periods p_i that are multiples of p_0, for each $i \geq 1$. The periods p_i satisfy the inequality*

$$p_{p_{i-1}} > p_{i-1} \tag{5}$$

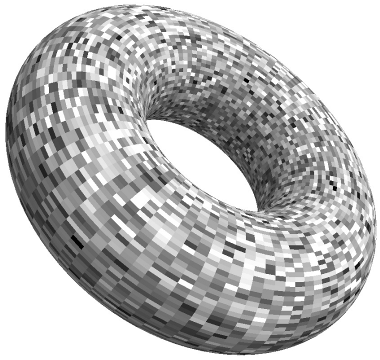
□

The inequalities (4) and (5) are much stronger for some quasigroups, i.e. it can happened $p_{s+1} > p_s$ for each s. In such a way we will obtain that $p_s \geq p_0 2^s$. So, starting from a string with period 1 we can produce strings with period $\geq 2^s$ by applying a transformation E or E'.

A PRSG are devices that produce random strings with elements of a set A. Since any device contains a determinism by itself, no one can guaranty that a theoretically ideal random string can be produced by such a device, and that is why only pseudo random strings can be produced. By Theorems 4 and 5 we can define a QPRSG as follows. Given a finite alphabet A of order $n > 3$ and a quasigroup operation $*$ on A, apply s times the transformation E (or E') on a periodical string on A. If the quasigroup $(A, *)$ is an exponential one, then the obtained output string will have an exponential period and the distribution of the letters, pairs of letters, ..., s-tuple of letters will be uniform. In such a way the output string will look very randomly. We can choose the number s of applications of the transformation E to be sufficiently large, in such a way a pseudo random strings with potentially infinite period and with uniform distribution of all tuples of letters will be obtained. It was shown that for some quasigroups in such a way obtained pseudo random strings passed all of the statistical tests for randomness we had on disposal [20]. Of course, the problem is how an exponential quasigroup to be obtained. It follows from the statistic presented in [2] and from our experiments presented here that there are sufficiently many exponential quasigroups and they can be effectively obtained. There is an effective way a random quasigroup of any order to be constructed (see [7]), and the random walk on torus can be applied for checking if the quasigroup is an exponential one.

Appendix 2: Graphical presentations

Here is a 'real' torus presentation at first:



Images of distributions of stops on torus of some classes of exponential quasigroups are given in Figure 1.

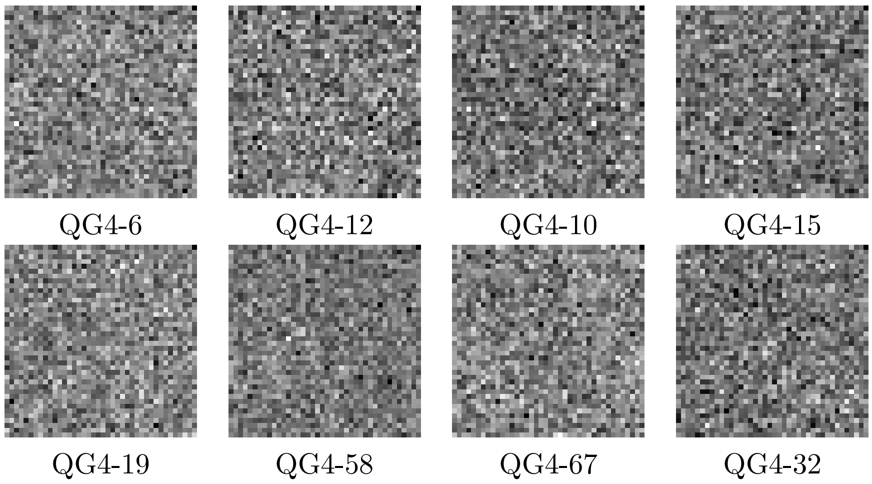


Fig. 1. Images of exponential quasigroups

Images representing the distribution of stops on some classes of linear quasigroups ar given in Figure 2.

Our experiments show that if there are internal symmetries in quasigroups then they are well visible from their images. We present 3 examples on Figure 4. We noticed that the permutation (0231) appears in some way in all of the given quasigroups. The appearance of the permutation (0231) is denoted on the quasigroups by the arrows $\rightarrow$, $\leftarrow$, $\uparrow$, $\downarrow$. Also, we note that the arrows are grouped two by two. This observation arise the following questions: If two quasigroups have the above property for some other permutation and other grouping of arrows will they have symmetrical images? Can

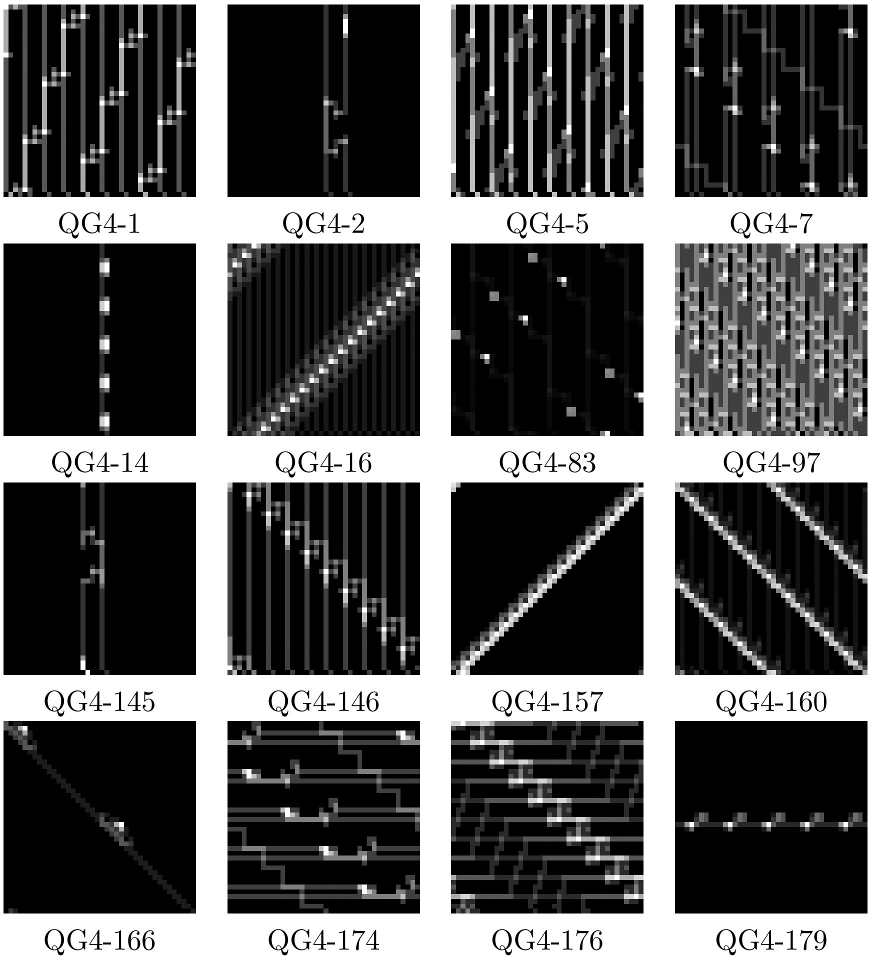


Fig. 2. Images of linear quasigroups I

we find quasigroups of higher order with this property? The example shows that the experimentally obtained results open questions that cannot be determined in another way.

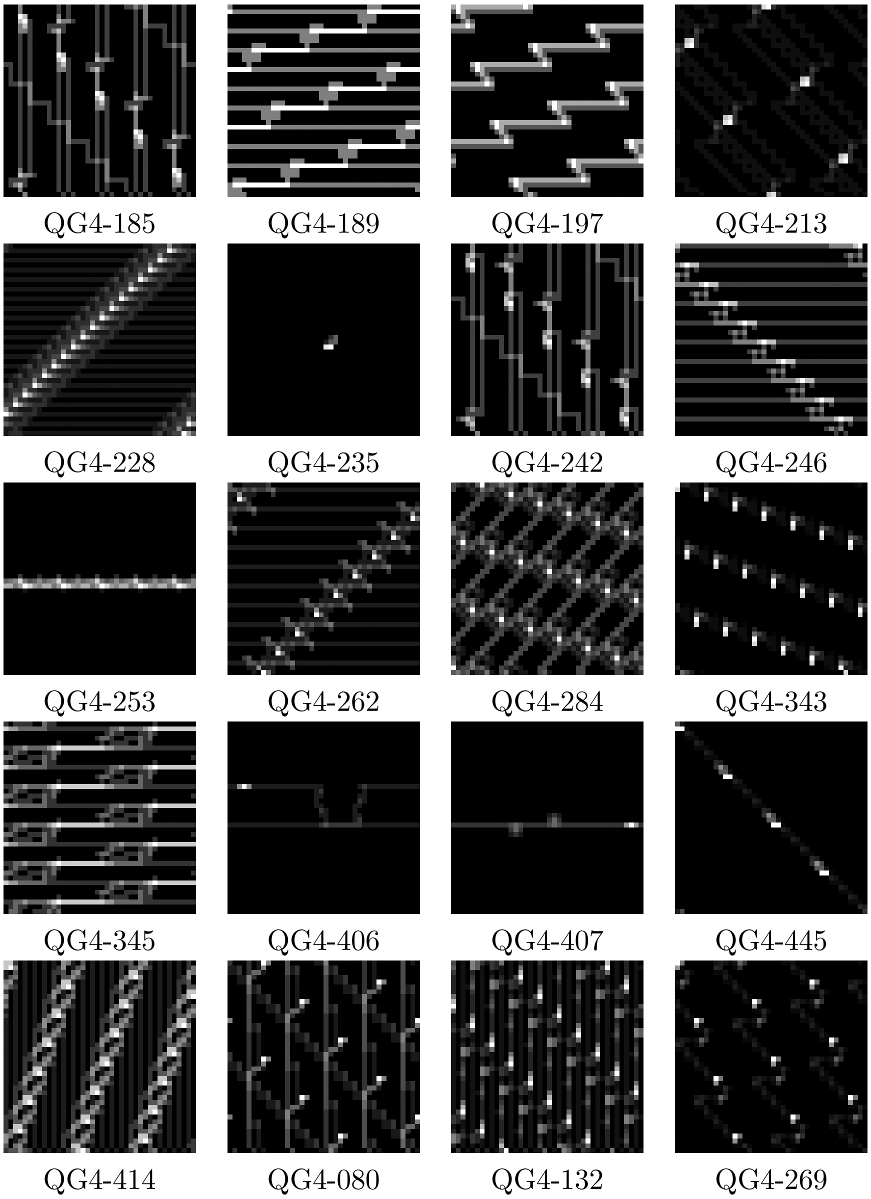
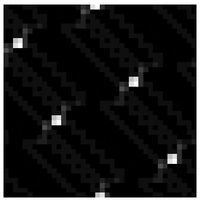
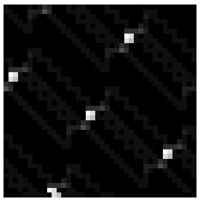


Fig. 2 (continued). Images of linear quasigroups I



QG4-213



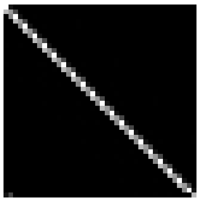
QG4-303

$$\begin{array}{cccc} 1 & 2 & 0 & 3 \\ 3 & 0 & 2 & 1 \\ 2 & 1 & 3 & 0 \\ 0 & 3 & 1 & 2 \\ \uparrow & \uparrow & \downarrow & \downarrow \end{array}$$

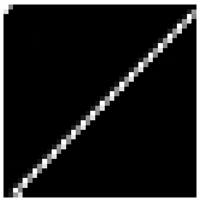
QG4-213

$$\begin{array}{ccccc} 2 & 0 & 1 & 3 & \leftarrow \\ 1 & 3 & 2 & 0 & \leftarrow \\ 3 & 1 & 0 & 2 & \rightarrow \\ 0 & 2 & 3 & 1 & \rightarrow \end{array}$$

QG4-303



QG4-293



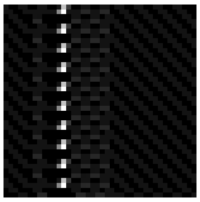
QG4-476

$$\begin{array}{ccccc} 2 & 0 & 1 & 3 & \leftarrow \\ 0 & 2 & 3 & 1 & \rightarrow \\ 3 & 1 & 0 & 2 & \rightarrow \\ 1 & 3 & 2 & 0 & \leftarrow \end{array}$$

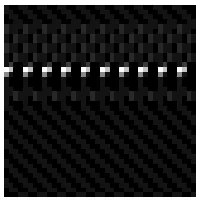
QG4-293

$$\begin{array}{cccc} 3 & 0 & 2 & 1 \\ 2 & 1 & 3 & 0 \\ 0 & 3 & 1 & 2 \\ 1 & 2 & 0 & 3 \\ \uparrow & \uparrow & \downarrow & \downarrow \end{array}$$

QG4-476



QG4-308



QG4-355

$$\begin{array}{ccccc} 2 & 0 & 1 & 3 & \leftarrow \\ 3 & 1 & 0 & 2 & \rightarrow \\ 1 & 3 & 2 & 0 & \leftarrow \\ 0 & 2 & 3 & 1 & \rightarrow \end{array}$$

QG4-308

$$\begin{array}{cccc} 2 & 1 & 0 & 3 \\ 3 & 0 & 1 & 2 \\ 1 & 2 & 3 & 0 \\ 0 & 3 & 2 & 1 \\ \downarrow & \downarrow & \uparrow & \uparrow \end{array}$$

QG4-355

Fig. 4. The internal symmetries of different quasigroups