Preface

The CALCULEMUS endeavour aims to explore the issues arising from the combination of automated mathematical computation and automated mathematical deduction, both broadly defined. This includes work aimed at more reliable and accurate computer algebra systems; more powerful and flexible automated deduction systems and at novel applications, for example to teaching. This is very much an exercise in bridge-building, uniting different mathematical foundations; different software engineering approaches; different user communities and different developer communities. For this reason, the Calculmus workshops co-locate in alternate years with either a Computer algebra conference or an automated reasoning conference.

The CALCULEMUS name is attached to a series of workshops, of which this is the most recent, to a European Union research programme funded under the IHP (Improving Human Potential) and to a loose interest group, including the IHP members. More information can be found at http://www.calculmus.net.

22 papers were submitted for the workshop, of which we have selected twelve regular papers and three system descriptions which are reprinted here, and a small number of others which, for various reasons are not included here, but will be presented orally or as posters at the workshop. In addition, we have, together with our host conference (IJCAR 2001), invited Prof. Doron Zeilberger to give an invited paper.

This year for the first time, there will be a special issue of the Journal of Symbolic Computation devoted to outstanding papers from, or relating to, Calculmus 2001, providing an archival publication route for Calculmus speakers. Submission will be required about three months after the meeting to allow publication before Calculmus 2002. Details will be available at or after the meeting.

Welcome to CALCULEMUS 2001!

June 2001

Steve Linton and Roberto Sebastiani

Co-chairs

Calculus 2001
Organization

Programme Committee

Andrew Adams       Reading
Alessandro Armando Genova
Clemens Ballarin  Karlsruhe
Jacques Calmet     Karlsruhe
Olga Caprotti      Eindhoven
Alessandro Cimatti  Trento
Edmund Clarke      CMU
James Davenport    Bath
John Harrison      Intel
Tudor Jebelean     RISC
Manfred Kerber     Birmingham
Michael Kohlhase    CMU
Steve Linton       St Andrews
Ursula Martin      St Andrews
Pierre-Etienne Moreau LORIA
Roberto Sebastiani Trento
Jörg Siekmann      Saarbrücken
Alan Smaill        Edinburgh
Carolyn Talcott    Stanford
Andrzej Trybulec   Białystok
Toby Walsh         York
Wolfgang Windsteiger RISC

Referees (other than PC members)

Christoph Benzmüller Mateja Jamnik Wolfgang Schreiner
Simon Colton        Koji Nakagawa Josef Schicho
Luca Compagna       Krzysztof Pramowski
Calulemus Trustees

Alessandro Armando  Jacques Calmet  Manfred Kerber  
Henk Barendregt  Arjeh Cohen  Michael Kohlhase  
Bruno Buchberger  Fausto Giunchiglia  Ursula Martin  
Alan Bundy  Tudor Jebelean  Jörg Siekmann  

Previous Meetings

16–19 March 1996, Rome  
July 1996, Workshop on IMACS  
18–20 November 1996 Dagstuhl, Germany  
28–30 April 1997, IRST, Italy  
24–26 September 1997, Edinburgh  
13–15 July 1998 Eindhoven  
5 July 1999, Trento  
6–7 August 2000, St Andrews
## Contents

### Full Papers

Sketches in Affine Geometry .................................................. 1  
*Norbert Preining*

The algebraic hierarchy of the FTA project ............................... 13  
*Herman Geuvers, Randy Pollack, Freek Wiedijk, Jan Zwanenburg*

On a Solution of Mutilated Checkerboard Problem Using the Theorema  
Set Theory Prover .............................................................. 28  
*Wolfgang Windsteiger*

An Agent-oriented Approach to Reasoning ................................. 48  
*Christoph Benzmüller¹, Mateja Jamnik², Manfred Kerber² and Volker Sorge¹*

Towards Mathematical Agents – Combining MATHWEB-SB and the LBA  
*Jürgen Zimmer, Alessandro Armando, Corrado Giomini*

Discovering Theorems using GöDEL: A Case Study ..................... 78  
*Johan Gjøbertus Frederik Beininfante*

A ‘Calculemus-approach’ to high-school math ? ........................ 92  
*Walter A. Neuper*

Computer Algebra meets Automated Theorem Proving: A Maple-PVS Interface ................................................................. 107  
*Martin Dunstan, (NAG Ltd.), Hanne Gottliebsen, Tom Kelsey, Ursula Martin (University of St Andrews)*

Equality in Computer Algebra and Beyond .............................. 120  
*James H. Davenport*

Extensions to Proof Planning for Generating Implied Constraints ...... 130  
*Alan M. Frisch, Ian Miguel and Toby Walsh*

Some hints for polynomials in the FOC project .......................... 142  
*S. Boulté, T. Hardin, R. Rioboo*

Incorporating Decision Procedures in Implicit Induction ............. 155  
*Alessandro Armando, Michaël Rusinowitch, Sorin Stratulat*

### System Descriptions

The AutoBayes Program Synthesis System — System Description — .... 168  
*Bernd Fischer¹, Thomas Pressburger¹, Grigore Roşu¹, Johann Schumann¹*
The **ACTIVEMath** Learning Environment System Description .......... 173
  *Paul Libbrecht, Erica Melis, Carsten Ullrich paul,melis,ullrich@ags.uni-sb.de*

System Description: Interface between Theorema and External Automated
Deduction Systems ................................................. 178
  *Temur Kutsia and Koji Nakagawa*
Sketches in Affine Geometry

Norbert Preining

Institute for Algebra and Computational Mathematics
University of Technology
Vienna, Austria
preining@logic.at

Abstract As shown in [BP01] sketches can be viewed as valid proving
tools in projective geometry where the number of primitive concepts is
limited to the notion of incidence, a point coincides with a line. The proof
uses Herbrand disjunctions to prove the equivalence between a formal-
ization of sketches and proofs in a formal calculus. In order to extend
this result to richer geometries the first step is to introduce the notion
of parallel to obtain affine geometry. Later on we will introduce more
and more concepts of Euclidean geometry like betweenness, congruence
or angle.
We will present a formalization of sketches in affine geometry which we
hope resembles the actual drawing of a sketch on a blackboard. Going
on we prove that any of these formal sketches can be translated into a
sketches in projective geometry and by this we can show that sketches can
be transformed into proofs. Finally the transformation from Herbrand
disjunctions, i.e. proofs, via projective sketches back to affine sketches
will prove that affine sketches and formal proofs are equivalent.

1 Introduction

Affine geometry can be considered as the basic of most natural geometries in the
sense that these geometries try to describe our surroundings. It is the geometry a
small child experiences while growing up. There is no measurement, no concept
distance, the only ideas we have are about incidence (“Mama is on the other
side of the street”) and parallelism (“I cannot reach her if I have to stay on this
side of the street”). This is affine geometry, incidence and parallelism, lines that
do not meet.

Ever since ancient civilizations like the chinese, arian or greek cultures started
to prove geometrical theorems there have been proofs based solely on sketches
and not on formal argumentation. With the beginning of strict formalization
of mathematics and the use of formal methods within mathematics, the way
sketches are used has changed the way they are used, because they could not
meet the requirements of current science for a valid proving method.

Sketches are known to be very useful in illustrating the facts of a proof and
in making the idea of a proof transparent. But sketches need not only be just a
hint, they can, in certain cases, be regarded as a proofs themselves. The purpose
of this paper is to extend results on sketches in projective geometry to affine
gometry.
1.1 Historic Examples

Proofs of geometric propositions are found in the earliest known mathematical texts in India and China as well as in Greece. Let us give as an example (taken from [vdW83]) a passage from the earliest Chinese text on astronomy and mathematics, the *Chou Pei Suan Ching*, which may be translated to "Arithmetical Classic of the Gnomon and the Circular Paths of Heaven". In this classical text from the Han-period a proof of the "Theorem of Pythagoras" is presented. The proof is only worked out for the (3, 4, 5) triangle, but the idea of the proof is perfectly general. Figure 1 and the translation are taken from [Nee]:

![Figure 1: The proof of the Pythagoras Theorem in the Chou Pei Suan Ching. From J. Needham: Science and Civilization in China, Vol. 3](image)

Thus, let us cut a rectangle (diagonally), and make the width 3 (units) wide, and the length 4 (units) long. The diagonal between the (two) corners will then be 5 (units) long. Now after drawing a square on this diagonal, circumscribe it by half-rectangles like that, which has been left outside, so as to form a (square) plate. Thus the (four) outer half-rectangles of width 3, length 4, and diagonal 5, together make two rectangles (of area 24); then (when this is subtracted from the square plate of area 49) the remainder is of area 25. This (process) is called "piling up the rectangles".

Today sketches have lost most of their importance as proving tools by themselves and are more or less only used to explain the ideas of a proof and to help understanding, but all the proofs have to be formalized in a strict sense. On the other hand there are some approaches to reintroduce sketches as proofs: For
example Grünbaum introduced a notion of provability in elementary geometry which is based on sketches made with Mathematica [Wol91]. Other approaches of proving with the help of sketches can be found in programs like Geometer's Sketchpad [Ste98] or DrGeo [Fer].

2 Sketches in Affine Geometry

Most of the proofs in affine geometry are illustrated by a sketch. This method of a graphical representation of possibly abstract facts is not only used in areas such as affine or projective geometry, but also in other fields such as algebra, analysis and sketches may even be used to support understanding in lectures on large ordinals, a highly abstract topic!

The difference between these sketches and the sketches used in affine geometry (and similar fields) is the fact that the proofs in affine geometry deal with geometric objects like Points and Lines, which are indeed objects we can imagine and draw on a piece of paper (which is not necessary true for large ordinals).

So the sketch in affine geometry has a more concrete task than only illustrating the facts, since it exhibits the incidences, which is the only predicate constant besides equality really needed in the normalization of affine geometry. It is a sort of proof by itself and so potentially interesting for a proof-theoretic analysis.

2.1 Introduction to Affine Geometry

What is Affine Geometry We will now give an axiomatization of affine geometry in a logical sense. The affine geometry deals, like the projective geometry, with points and lines. These two elements are primitives, which are not further defined. Only the axioms tell us about their properties.

Now let me begin with a definition of the affine geometry: There are two classes of objects, called Points and Lines, and two predicates (actually three, if we count the equality). One predicate defines a relation between Points and Lines, called Incidence, written \( P \cap g \) meaning the Point \( P \) incides with the line \( g \). The second predicate expresses a fact that two lines do not meet, called Parallelism, written \( g \parallel h \).

Furthermore we must give some axioms to express certain properties of Points and Lines and to specify the behavior of the incidence on Points and Lines:

- (AG1) For every two distinct Points there is one and only one Line, so that these two Points incide with this Line.
- (AG2) For a Point \( P \) and a line \( l \) such that \( P \not\in g \) there exists one and only one line \( m \) such that \( P \cap m \) and \( l \parallel m \).
- (AG3) There are three noncollinear Points.

---

1 We will use the expression "Point" (note the capital \( P \)) for the objects of affine geometry and "points" as usual for e.g. a point in a plane. The same applies to "Line" and "line".
Examples for Affine Planes

The Euclidean plane The Euclidean plane by itself is an affine plane, since Euclidean geometries just adds new concepts but do not change the basic axioms of affine geometry.

The minimal Affine Plane One of the basic properties of projective planes is the fact that there are four distinct Points, which is easy to prove. This is also the minimal number of points. If we can set up a relation of incidence on these Points such that the axioms (AG1) and (AG2) are satisfied, then we have a minimal affine plane. Fig. 2 defines such an incidence-table. In this table only the labelled points exist and there are no more lines then those drawn.

![Incidence Table for the minimal affine plane](image)

Some Consequences of the Axioms

- There exist four different points, no three collinear.
- There exist at least two different points on every line.
- Every point is on at least three different lines.

2.2 A Formalization of Sketches in Affine Geometry

In this part we give a formalization of the sketch in affine geometry and explain our motivation behind some of these concepts.

All Points and Lines are combined in the sets called $\tau_P$ and $\tau_L$, respectively.

**Definition 1 (Set of Terms over $\mathcal{C}$).** Let $\mathcal{C}$ be a set of constants of type $\tau_P$ or $\tau_L$, then $\mathcal{T}_n$ is inductively defined

- $\mathcal{T}_0(\mathcal{C}) = \mathcal{C}$
- $\mathcal{T}_{n+1}(\mathcal{C}) = \mathcal{T}_n(\mathcal{C}) \cup \{ [tu] : t, u \in \mathcal{T}_n(\mathcal{C}) ; t, u \in \tau_P \}$
  \[ \cup \{ (tu) : t, u \in \mathcal{T}_n(\mathcal{C}) ; t, u \in \tau_L \} \]
Definition 2 (Depth). The depth of a term \( t \) is defined as the number \( n \), at which \( t \) is added (or constructed) in the process given above.

To ensure consistency inside a set of starting objects, they must obey one rule, namely that if a compound term is in the set, than all of its subterms are also in the set. This is the reason for the next definition.

Definition 3 (admissible set of terms). Let \( \mathcal{M} \) be a subset of \( \mathcal{T}(C) \), \( C \) a set of constants, then \( \mathcal{M} \) is called admissible if it obeys the following rules:

- \((\forall [XY] \in \mathcal{M})(X,Y \in \mathcal{M})\)
- \((\forall (gh) \in \mathcal{M})(g,h \in \mathcal{M})\)

The idea is to define a set of Points, Lines and certain combinations of them (the intersection points and connection lines) and to let the sketch be a subset of all possible atomic formulas over these terms.

Definition 4 (Universe of Formulas). Let \( \mathcal{M} \) be an admissible termset and \( \mathcal{P} \) a set of predicates, then the universe of formulas over \( \mathcal{M} \) with regard to \( \mathcal{P} \) is defined as

\[
\mathcal{F}_\mathcal{P}(\mathcal{M}) = \{ P(t_1,t_2) : P \in \mathcal{P}; t_i \text{ of the right types} \}
\]

\( \mathcal{P} \) will only be \( \{ \| , = \} \) or \( \{ \| \} \). The set \( \mathcal{F}_\mathcal{P} \) contains all the possible positive statements which can be made over the termset \( \mathcal{M} \).

We wish to approximate real sketches as close as possible, and therefore we should not allow multiple instances of the same object, i.e. we require that a object (Point, Line) has a unique name and does not have different names in different parts. We require a proper state within our construction and therefore do not allow ambiguous information, which can arise from the following situation, called critical constellation:

Definition 5 (Critical Constellation). Let \( P \) and \( Q \) be terms in \( \tau_P \) and \( g \) and \( h \) terms in \( \tau_E \). Then we call the appearance of the following four formulas a critical constellation:

\[
\frac{P \ I \ g}{Q \ I \ h} \quad \frac{P \ I \ h}{Q \ I \ g}
\]

We will denote such critical constellations by \((P, Q; g, h)\).

Such a constellation is called critical, because from these four formulas it follows that either \( P = Q \) or \( g = h \) (or both), but we cannot determine which one of these alternatives without supplementary information (see fig. 3).

When constructing any sketch we start from some assumptions over a set of constants and then construct new objects and deduce new relations. From a proof-theoretic point of view these first assumptions are the left side of the deduced sequent, i.e. the assumptions from which you deduce the fact.

We now come to the definition of the sketch. We require a sketch to be a set describing all the incidences in the sketch\(^2\). But we also require that this

\(^2\) This one is on the paper!
subset is closed under trivial incidences, which means that if we talk about a Line which is the connection of Points, then we require that the trivial formulas express that these two Points lie on the corresponding Line.

Further we require that no critical constellations occur in a sketch. That arises from the fact that we wish that every geometric object is described only by one logical object, i.e. one term. Since a critical constellation implies the equality of two logical objects, which we cannot determine automatically, we want to exclude such cases.

Definition 6 (Sketch). Let $\mathcal{M}$ be an admissible termset over a set of constants $\mathcal{C}, \{A_0, B_0, C_0, [A_0 B_0], \ldots, [B_0 C_0]\} \subseteq \mathcal{M}$, let $\mathcal{E}$ be a subset of $\mathcal{FU}_{\{\neq\}}(\mathcal{M}) \cup \mathcal{FU}_{\{=\}}(\mathcal{M})$ with $A_0 \neq B_0, \ldots, B_0 \neq C_0, A_0 \neq [B_0 C_0], \ldots, B_0 \neq [A_0 C_0] \in \mathcal{E}$, let $Q$ be a set of equalities and let the triple $(\mathcal{M}, \mathcal{E}, Q)$ obey the following requirements:

$(\forall X, Y \in \mathcal{M}, \tau_P)([XY] \in \mathcal{M} \supset (X \ I [XY]) \in \mathcal{E} \land (Y \ I [XY]) \in \mathcal{E})$

$(\forall g, h \in \mathcal{M}, \tau_P)((gh) \in \mathcal{M} \supset ((gh) \ I g) \in \mathcal{E} \land ((gh) \ I h) \in \mathcal{E})$ (S.1)

$(\neg \exists x, y \in \mathcal{M})(P(x, y) \in \mathcal{E} \land \neg P(x, y) \in \mathcal{E})$

$(\forall x \in \mathcal{M})(x \neq x) \notin \mathcal{E}$ (S.2)

there are no critical constellations in $\mathcal{E}$ (S.3)

$(\forall x \in \mathcal{M})(x = x) \in \mathcal{Q}$ (S.4)

Then we call the triple $\mathcal{S} = (\mathcal{M}, \mathcal{E}, \mathcal{Q})$ a sketch.

We will call the violation of S.2 a direct contradiction.

A small example should aid understanding of the concepts:

In the sketch depicted in fig. 4 the different sets are (where the incidences of the constants are lost!):

$\mathcal{C} = \{P, Q, R, X, g\}$

$\mathcal{M} = \{P, Q, R, X, g, [RQ]\}$

$\mathcal{FU}_{\{\neq\}} = \{P \ I g, Q \ I g, R \ I g, X \ I g,$

$\quad P \ I [RQ], Q \ I [RQ], R \ I [RQ], X \ I [RQ], P = Q, P = R, P = X, Q = R, Q = X, R = X,$

$\quad g = [RQ]\}$

$\mathcal{E} = \{Q \ I [RQ], R \ I [RQ], X \ I g, P \ I g, Q \ I g, R \ I g, P \ I [QR], X \ I [QR]\}$

$\quad P = Q, P = R, P = X, Q = R, Q = X, R = X,$

$\quad g = [RQ]\}$

Figure 3: The two solutions for a critical constellation
A few words on notation: If we are writing expressions like $P \in S, (P \parallel g) \in S, (P \parallel h) \in S, \ldots$, then $P \in M; (P \parallel g), (P \parallel h) \in \mathcal{E}$, respectively is meant. Any other similar expression has to be interpreted accordingly.

Why should the set $\mathcal{E}$ only contain a subset of $\mathcal{F}(\{z, \|\})$ and not of $\mathcal{F}(\{z, \|, -\})$? The reason is, that in a sketch every geometric object should have one and only one name and should also be described by one logical object. The same idea lies behind the introduction of the concept of the critical constellation.

Note that one sketch is only one stage in the process of a construction, which starting from some initial assumptions forming a sketch deduces more and more facts and so constructs more and more complex sketches.

The set $Q$ in the definition of the sketch initially was absent, but investigations in the equality of proofs and constructions showed that this set is important for the proof, although it is not used in the sketch. This depends on the usage of the equality: in the sketch it is a strict one, i.e., there is only one name for an object allowed, while in a proof you can use one name at one time and a different one subsequently. In the sketch, as we will see later, there is not a local substitution of a term, but a global, therefore only one name is “actual” at a time for an object. But if we want to translate a proof into a construction, which is one of the aims of this work, we need informations on all the name-changes that are possible.

2.3 Actions on Sketches

Till now a sketch is only a static concept, nothing could happen, you cannot “construct”. So we want to give some actions on a sketch, which construct a new sketch with more information. The new sketch may not have the properties S.1–S.3, but it must be a semisketch:

**Definition 7 (Semisketch).** A semisketch is a sketch that need not obey to S.2 and S.3.

These actions should correspond to similar actions in real world, i.e. actions taken when one draws a sketch. After these actions are defined we can explain what we mean by a construction in this calculus for construction.
The following list defines the allowed actions and what controls have to be executed. The following list describes the changes that have to be done on the quadruple of a sketch when we carry out the corresponding action.

In the following listing we will use the function closure$(Q)$ on a set of equalities $Q$. This function deduces all equalities which are consequences of the set $Q$. This is a relatively easy computation. If we have $Q = \{x = y, y = z, x = y, y = z\}$, then the procedure returns $Q \cup \{x = z\}$. This function is used to update the set $Q$ of a sketch after a substitution.

**Joining of two Points $X, Y$; Symbol:** $[XY]$: $\mathcal{M}' = \mathcal{M} + [XY], \mathcal{E}' = \mathcal{E} + \{X \mathcal{I} [XY], Y \mathcal{I} [XY]\}, Q' = Q + ([XY] = [XY])$. The requirements (S.1) and (S.4) are fulfilled since the necessary formulas are added to $\mathcal{E}$ and $Q$. This action can produce a semisketch from a sketch.

**Intersection of two Lines $g, h$; Symbol** $(gh)$: Dual to the joining of two points, but this action will only be allowed if in $\mathcal{E}$ there is $g \parallel h$.

**Assuming a new Object $C$ in general position, Symbol** $\{C\}$: $\mathcal{M}' = \mathcal{M} + C, \mathcal{E}' = \mathcal{E}, Q' = Q + (C = C)$. That $\mathcal{S}'$ is a sketch is trivial, since $C$ is a completely new constant. $C$ must be a constant of type $T_\parallel$ or $T_\perp$.

**Giving the Line $[XY]$ a new name $g := [XY]$; Symbol** $g := [XY]$:

$\mathcal{M}' = \mathcal{M}([XY]/g), \mathcal{E}' = \mathcal{E}([XY]/g), Q' = Q([XY]/g)$. $\mathcal{S}'$ is a sketch since this operation is only a name-change. Note that $g$ must not be in $\mathcal{M}$.

**Drawing a line parallel to $g$ through $P$; Symbol** $h := \parallel (g, P)$: $\mathcal{M}' = \mathcal{M} + h, \mathcal{E}' = \mathcal{E} \cup \{g \parallel h, P \mathcal{I} h\}, Q' = Q + (h = h)$.

**Identifying the Point $(gh)$ a new name $P := (gh)$; Symbol** $P := (gh)$

Dual to giving an intersection-point a name. Note that $P$ must not be in $\mathcal{M}$.

**Identifying two Points $u$ and $t$; Symbol** $u = t \mathcal{M}' = \mathcal{M} \setminus \{u\}, \mathcal{E}' = \mathcal{E}[u/t], Q' = \text{closure}(Q \cup \{u = t\})$. Note that the set $Q'$ can contain terms $t$ not in $\mathcal{M}'$. This action can produce a semisketch from a sketch.

**Identifying two Lines $l$ and $m$; Symbol** $l = m$

Dual to identifying two Points.

**Using a “Lemma”:** Adding $t \mathcal{I} u$; Symbol $t \mathcal{I} u \mathcal{M}' = \mathcal{M}, \mathcal{E}' = \mathcal{E} + (t \mathcal{I} u), Q' = Q$. This action can produce a semisketch from a sketch.

**Adding a negative literal $t \nmid u$; Symbol** $t \nmid u \mathcal{M}' = \mathcal{M}, \mathcal{E}' = \mathcal{E} + (t \nmid u), Q' = Q$.

**Adding a negative literal $t \neq u$; Symbol** $t \neq u \mathcal{M}' = \mathcal{M}, \mathcal{E}' = \mathcal{E} + (t \neq u), Q' = Q$.

To deduce a fact with sketches we connect the concept of the sketch and the concept of the actions into a new concept called construction. This construction will deduce the facts.

**Definition 8 (Construction).** A construction is a rooted and directed tree with a semisketch attached to each node and an action attached to each vertex and satisfying the following conditions: If a vertex with action $A$ leads from node $N_1$ to node $N_2$, then $N_2$ is obtained from $N_1$ by carrying out the action on $N_1$. If from a node $N$ there is a vertex labeled ...
- with \([XY]\) or \((gh)\), then \(X \neq Y\) or \(\{g \neq h, g \parallel h\}\) is contained in \(\mathcal{E}^N\).
- with \(h :=\| (g, P), \{C\}, g := [XY], P := (gh)\), then there is no other vertex from \(N\).
- with \(P \nparallel g, X = Y, g \parallel h\), then there is exactly one other vertex labeled with the negation of the other formula. This is called a case-distinction.

Furthermore if \(\mathcal{E}\) attached to a node...
- yields a direct contradiction, then it has no successor;
- contains formulas \(P \nparallel g, P \nparallel h, g \parallel h\) and \(g \neq h\) then there is no successor; the node is contradictory.
- is a semi-sketch but not a sketch, i.e. that there are critical constellations, let \((P, Q; g, h)\) be one of them, then there are exactly two successors, one labeled with the action \(P = Q\) and one labeled with the action \(g = h\).

What is deduced by a construction: A formula is true when it is true in all the models of the given calculus. The distinct models in a construction are achieved by case-distinctions. So if a formula should be deduced by a construction, it must be in all the leafs of the tree. But since some leaves end with contradictions and from the logical principle “ex falso quodlibet” we only require that a formula, which should be deduced, has to be in all leaves which are not contradictory.

We also have to pay attention to the way a construction handles identities. Since in a construction an identity is carried out in the way that all occurrences of one term are substituted for the other, we not only prove an atomic formula, but also all the formulas which are variants with respect to the corresponding set \(Q\). This notion will now be defined.

**Definition 9.** Two atomic formulas \(P(t_1, u_1)\) and \(P(t_2, u_2)\) are said to be equivalent with respect to \(Q\), where \(Q\) is a set of equalities, in symbols \(P(t_1, u_1) \equiv_{Q^N} P(t_2, u_2)\), when \((t_1 = t_2), (u_1 = u_2) \in Q^N\) (or the symmetric one).

Now we can define the notion of what a construction deduces:

**Definition 10.** A construction deduces a set of atomic formulas \(\Delta\) iff for all \(A \in \Delta\) there is a not contradictory leaf, where either \(A \in Q_N\) or \((\exists B \in \mathcal{E}(N))A \equiv_{Q[N]} B\).

The meaning of this definition is that if a construction deduces \(\Delta\) then the disjunction of all formulas in \(\Delta\) is proved by this construction.
2.4 An example for an affine construction

We will prove the following sentence of affine geometry with a construction:

\((\neq (l, m, n) \land m \parallel n \land P \not\perp l \land P \not\perp m) \supset (\exists Q)(Q \not\perp l \land Q \not\perp n)\)

I.e. if two lines \((m, n)\) are parallel and another line \((l)\) intersects with one of them, then it intersects also with the other one.

The construction is given in fig. 6. First the assumptions are build up by simple case distinction, this is an automatic process, then the proof by sketch follows closely any other proof by distinction whether \(l \parallel n\) or not. In case it is parallel node 9 is reached and closed because it yields a contradiction \((P \not\perp l, P \not\perp m, l \parallel m, l \neq m)\). If they are not parallel we can construct the intersection point \((n)\) and are finished. Therefore the construction proves the above formula (if the implication is transformed into a disjunction).

![Diagram](image_url)

Figure 6: An affine construction

3 Translation of Sketches to Proofs and back

We will explain how to go from sketches in affine geometry as defined above to proofs in a formal calculus and how to obtain a set of sketches that prove the same as a proof in a formal calculus.

The translation is done by transforming a given affine construction into a projective one, which in turn can be translated into a proof according to [BP01]. A similar route is used for the reverse direction: If we have a proof of a formula in affine geometry we transform it into a formula in projective geometry, build the corresponding projective sketch and translate it into an affine sketch.

Both translations are based on the well known fact that you can complete an affine geometry with one line at infinity to a projective geometry, and that you can strip down a projective geometry by deletion of one line to an affine...
geometry. In these cases parallel means that the intersection point of two lines incides with the taken out or added line.

It is easy to prove the axioms of affine geometry are valid in this new structure: Axiom (AG1) is the same in projective geometry, for axiom (AG2) just define \( m := [P(tu)] \). It is obvious that \( P \parallel m \) and that this line is unique. Finally the last axiom (AG3) is a trivial consequence from the last axiom of projective geometry.

On the other hand we can extend any affine geometry by adding a line \( u \) which holds all the meeting points of parallel lines to a projective geometry.

**Lemma 1.** By adding/deleting one Line to/from an affine/projective geometry we obtain a projective/affine geometry.

So we can define parallel by incidence and the added line as

\[
g \parallel h := (g = h) \lor ((gh) I u).
\]

**Lemma 2.** Any construction in affine geometry can be translated into a construction in projective geometry.

**Proof:** First note that the actions are the same in both formalizations, only the action of drawing a line parallel to a given line through a given point is added. But this action can be emulated by constructing the term \( h := [P(gu)] \). Moreover all the instances of \( g \parallel h \) in the sets \( \mathcal{E} \) must be changed to \((gh) I u \). With these two substitutions the transformation into construction in projective geometries is done and the transformation into proofs is done according to [BP01].

Now we show that we can do the reverse process, too. We start with a formula of affine geometry. Then we rewrite the axioms and add the line \( u \) to obtain a formula in projective geometry. This one can be proven by sketches (again according to [BP01]). Finally we have to transform the sketches in projective geometry back into affine geometry.

**Lemma 3.** The construction in projective geometry of a Herbrand disjunction of an affine geometry formula can be transformed into a construction in affine geometry.

**Proof:** Due to the fact that the Herbrand disjunction of the formula is also a affine formula, the translation of the Herbrand disjunction into projective geometry only contains \( u \) in formulas like \( g = h \lor (gh) I u \). In the projective construction we exchange the following parts: This changes the actions and therefore the sets \( \mathcal{E} \) are changed, too. This way all the occurences of \( u \) within the construction are cancelled and we obtain a affine construction. \( \square \)

As a consequence of the above lemmata we can state

**Theorem 1.** Affine sketches and proofs are equivalent in the sense that any proof of an affine sentence can be translated into a (set of) construction(s) which deduce the same sentence, and any affine construction can be transformed into a proof.
4 Closing Comments

We have proven that sketches in affine geometry can be viewed as proofs by themselves. Some consequences from the properties of projective sketches translate into affine geometry. Most significant is the fact that sketches are not constructive in mathematical sense, because of the concept of *general position* which comes down to case distinction within the sketches.

This paper is the first one on the way to extend results on projective geometry to Euclidean geometry.

References


The algebraic hierarchy of the FTA project

Herman Geuvers, Randy Pollack, Freek Wiedijk, Jan Zwanenburg
University of Nijmegen

Abstract We describe a framework for algebraic expressions for the proof assistant Coq. This framework has been developed as part of the FTA project in Nijmegen, in which a complete proof of the fundamental theorem of algebra has been formalized in Coq. The algebraic framework that is described here is both abstract and structured. We apply a combination of record types, coercive subtyping and implicit arguments.

The algebraic framework contains a full development of the real and complex numbers and of the rings of polynomials over these fields. The framework is constructive. It does not use anything apart from the Coq logic. The framework has been successfully used to formalize non-trivial mathematics as part of the FTA project.

1 Introduction

1.1 Background

When we started working on the FTA project in the proof tool Coq [2] (see Sect. 2 for an overview of this project) we needed an algebraic framework for it. There were several requirements for such a framework:

- We wanted to translate a constructive proof of the fundamental theorem of algebra, so needed support for constructive algebra.
- We wanted to reason about real numbers and polynomials over the reals.
- We wanted to use an abstract presentation of the real numbers that could be instantiated with various constructions in the future. In intensional type theory, such as Coq, this requires the use of setoids, which are a type and an equivalence relation packaged together.

No algebraic developments in Coq at that time could be a base for our work, so we decided to develop our own algebraic hierarchy for Coq. This would be a constructive theory of the real numbers and beyond, built on a pervasive notion of constructive setoid.

1.2 Approach

We did not want our mathematics to be dependent on specific representations of the various algebraic structures, so we decided to use an axiomatic approach. Instead of defining the real numbers as, e.g., Cauchy sequences or Dedekind cuts,
we defined a notion of real number structure of which both these constructions would be an instance. For any ‘implementation’ of this notion all of our theory would immediately be available.

Constructive mathematics is supposed to have computational content, but a proof of FTA based on an abstract real number structure, although using only constructive principles, cannot compute the number that the theorem claims exists. (Because the real number structure is like an abstract datatype specification without an implementation.) However, instantiating the abstract proof with any construction of the real number structure gives the proof full computational content.

1.3 Related work

Other algebraic hierarchies are similar to ours. However they often have not been used in a large proof development. Our framework has been written and optimized for real-life use (to prove the fundamental theorem of algebra) and not as a toy exercise in abstract mathematics.

Paul Jackson, in his Ph.D. thesis [10], presents a constructive development of algebra in Nuprl and uses it to prove some results in abstract algebra. Nuprl, like Coq, is based on type theory, but it uses an extensional equality. This makes several constructions easier (e.g. quotienting), but renders type checking undecidable. As Nuprl does not have coercive subtyping, there is in general no inheritance between structures.

Loïc Pottier implemented a classical algebraic hierarchy for Coq. This hierarchy is more elaborate than ours and mimics the hierarchy of the Axiom computer algebra system [11]. This hierarchy has been presented in conferences (Calculemus ’98, Types ’99), but there is nothing about it in the proceedings. All that exists is a brief web page in French [18].

A constructive algebraic hierarchy for Lego in the work of Anthony Bailey has only been presented in his Ph.D. thesis [1], which is not readily available. (Bailey left research and has no web page.)

Many provers have a formalization of the real numbers. Often such a formalization is ‘flat’ – the type of the real numbers is not an instance of a notion of ‘field’ – but some are part of an algebraic hierarchy.

A flat development of the real numbers has been added to the Coq library by Micaela Mayero [5]. This development only gives axioms for the real numbers and does not implement a specific model of it. Again this work is not constructive.

The Mizar system [15, 20] has both flat real numbers as well as an algebraic hierarchy. Interestingly, it is the flat real numbers that is used in most proofs that need the real numbers. The Mizar algebraic hierarchy is scattered throughout the Mizar library and has not been designed by one person. For instance the type group is in an article by Michał Muzalewski and Wojciech Skaba, the type Ring is in an article by Henryk Oryszczyszyn and Krzysztof Prażmowski and the type Field is in an article by Eugeniusz Kusak, Wojciech Leonczuk and Michał Muzalewski.
1.4 Contribution

We have built a library for doing algebra and analysis in Coq. It is completely self-contained and completely constructive. That is, it uses the Coq logic (the calculus of inductive constructions) and contains no axioms. It contains a construction of the real numbers as Cauchy sequences, so despite the fact that our theory is axiomatic in spirit we also give a specific representation of the field of the real numbers.

Subjects treated in our framework are:

- groups, rings and fields
- finite sums
- polynomials
- finite dimensional vector spaces
- the real and complex numbers
- real valued functions and continuity
- the intermediate value theorem
- real and complex roots

All together our Coq code consists of approximately 35000 lines or 860 kilobytes.

The theory that we developed contains many ‘calculation lemmas’. These are lemmas that give a calculation rule used to manipulate algebraic expressions. We created an automatic way to keep track of these lemmas in the form of a \LaTeX document, in the spirit of ‘literate programming’. Also we developed specific reasoning tools to make it easier to reason inside our framework without having to know the names of the lemmas [7].

1.5 Outline of the paper

In Sect. 2 an overview of the FTA project is presented. Sect. 3 gives the notion of constructive setoid. Building on this, Sect. 4 presents the algebraic hierarchy. Sect. 5 develops our approach to representing the inheritance of common notions in the hierarchy. Sect. 6 describes the way our framework models partial functions such as division. For this we introduce the notion of \textit{subsetoid}. Finally Sect. 7 discusses the syntax of expressions and the complexity of the underlying Coq terms.

In order to understand this paper one must have some basic familiarity with the Coq system [2].

2 The Fundamental Theorem of Algebra project

In 1999 and 2000 in the group of Henk Barendregt at the University of Nijmegen, a proof of the fundamental theorem of algebra was formalized in the Coq proof assistant [2]. This work was done in the spirit of the QED Manifesto [4].

The proof of the FTA project is constructive. Before the FTA project had been finished, a classical proof of the fundamental theorem of algebra had already
been formalized by Robert Milewski in Mizar [14] and by John Harrison in HOL [9].

The people involved with the FTA formalization were:

- Herman Geuvers
- Randy Pollack
- Freek Wiedijk
- Jan Zwanenburg
- Milad Niqui
- Henk Barendregt

The first four people designed the algebraic hierarchy that is in this paper. Milad Niqui wrote the ‘implementation’ of the real numbers in this framework [6]. All together the formalization of the proof of the fundamental theorem of algebra took between three and four man-years.

The fundamental theorem of algebra states that every non-constant polynomial over the complex numbers has a zero. In other words it says that the field of the complex numbers is algebraically closed. The proof that was translated to Coq was a constructive proof by Hellmuth and Martin Kneser [12, 8]. Because of its constructive nature we could keep the formalization free of axioms. All that we needed was already present in the Coq logic, the calculus of inductive constructions.

The final statement that we proved in the formalization was:

```
Lemma fta :
  (f:(c.poly_cring CC))(nonConst ? f)->(EX z | f!z[]=Zero).
```

This says that for every non constant polynomial \( f \) over the complex numbers there exists some complex number \( z \) such that \( f(z) = 0 \).

3 Setoids, constructive setoids and setoid functions

Coq is an intensional type theory. A notion called \textit{Leibniz equality} is definable; it is the smallest reflexive relation. In a context with no assumptions about Leibniz equality, it is equivalent to the meta theoretic (intensional) \textit{definitional equality}. On concrete constructions, such as natural numbers, where equality and identity coincide, Leibniz equality coincides with the definable structural identity. However, in abstract mathematics, this is not a very useful relation. For instance consider the representation of real numbers as Cauchy sequences. Two different Cauchy sequences (i.e. intensionally distinguished by Leibniz equality) can represent the same (extensionally) real number. If we were to assume axioms restricting Leibniz equality to behave as intended for an abstract real number structure, it would be impossible to implement that structure with a construction.

We want to make a \textit{quotient type} by dividing out the equivalence relation ‘represent the same real’. The solution is to work with \textit{setoids} instead of raw types. A setoid is a type together with an equivalence relation on it. (This
equivalence, called setoid equality, or book equality, is written \(x =_p y\) in our use of Coq notation.) Quotienting a setoid is achieved by changing its equivalence relation.

In constructive mathematics there is a little more to say, as the notion of of apartness (written \(x \neq y\)) is more fundamental than the notion of equality. No amount of information can show that concretely presented real numbers are equal. For example, consider real numbers, \(x\) and \(y\), presented as Cauchy sequences: however many terms of of \(x\) and \(y\) we have examined and found to be equal, we can not be sure that some later terms will not distinguish \(x\) from \(y\). However, after some number of terms we may see that \(x\) and \(y\) are so far apart that, being Cauchy sequences, they cannot represent the same real. Two objects are apart if we 'positively' know (have evidence) that they are different. For instance a real number \(x\) is apart from 0 only if we can give some natural number \(n\) such that we know that \(|x| > 1/n\).

Packaging a carrier set with an equivalence relation and an apartness relation, we get the notion of constructive setoid, called CSetoid in our framework. Apartness in such a CSetoid is written \(x # y\). In Coq we define constructive setoid as a record type:

```
Record CSetoid : Type :=
{ cs_crr :> Set;
  cs_eq : (relation cs_crr);
  cs_ap : (relation cs_crr);
  cs_proof : (is_CSetoid cs_crr cs_eq cs_ap)
  }.
```

A record type in Coq consists of labelled tuples, where the type of a field may be dependent on other fields. A term of type CSetoid is a tuple \((A, R1, R2, p)\), with \(A: Set, R1: (relation A), R2: (relation A)\) and \(p: (is_CSetoid A R1 R2)\). (\(p\) is a proof that \(A, R1, R2\) have property \(is_CSetoid\).) The labels allow projection of a specific field of the tuple: if \(S: CSetoid\), then \((cs_crr S): Set\) and \((cs_eq S): (relation (cs_crr S))\). As a matter of fact, the projection \(cs_crr\) does not have to be written, because \(cs_crr\) is declared (by the annotation :>) to be a coercion function. This means that the type checker will insert this function when necessary, driven by typechecking. For example \(S\) is not a type, and cannot have inhabitants, but if we declare a variable \(x:S\), the type checker implicitly forms the correct declaration \(x: (cs_crr S)\). This captures the common mathematical usage of confusing a structure with its carrier.

In the above definition, \(is_CSetoid\) is a predicate, the conjunction of the defining properties of a constructive setoid. It is itself defined as a record:

```
Record is_CSetoid [A:Set; eq,ap:(relation A)] : Prop :=
{ ax_ap_irreflexive : (irreflexive A ap);
  ax_ap_symmetric : (symmetric A ap);
  ax_ap_cotransitive : (cotransitive A ap);
  ax_ap_tight : (tight_apart A eq ap)
  }.
```
This says that a constructive setoid is a tuple \( \langle A, \approx, \# \rangle \) that satisfies, for all \( x, y \) and \( z \) in \( A \):

- apartness is irreflexive: \( \neg(x \neq x) \)
- apartness is symmetric: \( x \neq y \rightarrow y \neq x \)
- apartness is cotransitive: \( x \neq y \rightarrow x \neq z \lor z \neq y \)
- apartness is tight: \( \neg(x \neq y) \leftrightarrow x \approx y \)

Because of the property of tightness we could have defined equality in terms of apartness, rather than carrying it in our setoid structure. Because equality seems such an important notion we didn’t do this.

The field cs_eq of a CSetoid record is the function that represents the equality. Hence the Coq expression:

\[
(\text{cs\_eq} \ S \ x \ y)
\]

represents \( x \approx y \) in \( S \). Since Coq can determine the argument \( S \) from the types of \( x \) and \( y \), we can define an operator \( [=] \) such that \( x [=] y \) is a shorthand for \( (\text{cs\_eq} \ S \ x \ y) \). Similarly we can define \( x \neq y \) as an abbreviation of \( (\text{cs\_ap} \ S \ x \ y) \).

The notion setoid allows intensional formalization of quotient. A function \( f \) only induces a corresponding function on a quotient of its domain when it respects the equivalence \( \approx \) that is being divided out:

\[
x \approx y \rightarrow f(x) \approx f(y)
\]

This is called weak extensionality or well-definedness of the function. It is defined in the Coq formalization as:

**Definition** fun\_well\_def \( [f:S1\rightarrow S2] : \text{Prop} := \)

\[
(x,y:S1) (x[=]y) \rightarrow ((f x)[=](f y)).
\]

Constructively, as apartness is more fundamental than equality, so the property of strong extensionality

**Definition** fun\_strong\_ext \( [f:S1\rightarrow S2] : \text{Prop} := \)

\[
(x,y:S1) ((f x)[#](f y)) \rightarrow (x[#]y).
\]

is more fundamental than well-definedness. It is easily shown that strong extensionality implies well-definedness. The properties of well-definedness and strong extensionality are also defined for relations.

There naturally occur functions that are not well-defined in the above sense. For instance the \( n \)-th root \( \sqrt[n]{x} \) in the complex plane; it is possible to define the \( n \)-th root of a complex number, but different representations of the same complex number will sometimes have different complex roots (not just different representations of the same complex root). So although the \( n \)-th root function can be defined it does not respect the setoid equality. Thus it is necessary to distinguish between functions, which are ‘just’ terms \( f \) of functional type \( S1\rightarrow S2 \),

---

18 Herman Geuvers, Randy Pollack, Freek Wiedijk, Jan Zwanenburg
and \textit{setoid functions}, which should also be strongly extensional (and hence respect the equality).\footnote{Bishop [3] calls these \textit{operations} and \textit{functions} respectively.} Note that the non-well-definedness of some functions is unavoidable and is not caused by the constructivity; classically, the $n$-th root is non-continuous as a function from $\mathbb{C}$ to $\mathbb{C}$. (It can be made continuous by mapping to \textit{sets} of roots, with an appropriate topology on those sets.)

4 \textbf{Algebraic structures and coercive subtyping}

We defined a number of types in Coq representing algebraic structures of which the carriers are constructive setoids. Each algebraic type is defined in terms of the previous one.

\begin{quote}
\begin{tabular}{ll}
CSetoid & constructive setoids \\
CSemi.grp & semi-groups \\
CMonoid & commutative monoids \\
CGroup & groups \\
CRing & rings \\
CField & fields \\
COrdField & ordered fields \\
CReals & 'real number structures'
\end{tabular}
\end{quote}

We lack the space to present the definitions of these types in detail, and refer the interested reader to the FTA files which are available on the web at URL \texttt{<http://www.cs.kun.nl/~freek/fta/>}. The definitions of the types of the algebraic hierarchy have been extracted in the file \texttt{Spec_CReals.v}.

In this paper we only present the definition of the type of rings in terms of the type of groups; the other definitions follow the same pattern. The type of rings is defined:

\begin{verbatim}
Record CRing : Type :=
{ cr_crr :> CGroup; 
  cr_one : cr_crr; 
  cr_mult : (CSetoid_bin_op cr_crr); 
  cr_proof : (is_CRing cr_crr cr_one cr_mult) 
}.
\end{verbatim}

The function \texttt{cr_crr} that gives the underlying group is a coercion (indicated by the annotation :>) which Coq can silently insert, as explained above (Sect. 3). So the type \texttt{CRing} is a coercive subtype of \texttt{CGroup}. For details of coercive subtyping in Coq see [19, 2].

The multiplication operation of a ring is a setoid function: it respects setoid equality. We have types for such setoid functions called \texttt{CSetoid_bin_fun} and \texttt{CSetoid_bin_op} (the second is a specialized case of the first in which the domain and range setoids are the same).

The defining property \texttt{is_CRing} is:
Record is_CRing
   [G:Group; one:G; mult:(CSetoid_bin_op G)] : Prop :=
   { ax_mult_assoc : (Associative mult);
     ax_mult_mon :
       (is_CM_monoid (Build_CSemigroup G one mult ax_mult_assoc));
     ax_dist : (Distributive mult (csg_op G));
     ax_non_triv : one[≠]Zero
   }.

This completes the definition of rings in terms of groups.
The general scheme of defining an algebraic structure B in terms of an algebraic structure A is:

Record BName : Type :=
   { b_crr => AName;
     b_opName1 : σ1;
     ...
     b_opName_n : σ_n;
     b_proof : (is_BName b_crr b_opName1 ... b_opName_n) }

Record is_BName
   [A:AName; opName1:σ1; ...; opName_n:σ_n] : Prop :=
   { ax_propName1 : P1;
     ...
     ax_propName_m : P_m }

Note that BName is not a structural subtype of AName in the sense of having at least all the fields of AName. Instead AName occurs as a field in BName. (Records in Coq are right associative and not extensible, in the classification of [17].) However BName is a subtype of AName by the coercion b_crr, so that wherever an AName is expected, a BName can be used instead.

5 Three ways to classify addition

We will now compare three ways to treat addition in Coq. The first is the way it is done in the standard Coq library, the second is a bridge to the third, which is the way it is done in the algebraic hierarchy.

5.1 The standard Coq library: separate additions

In the naive approach one defines an addition for every new type with addition that is introduced. In the Coq standard library there are three different additions for the natural numbers, the integers and the reals:
plus : nat->nat->nat
Zplus : Z->Z->Z
Rplus : R->R->R

The properties of these additions must be developed (or assumed) from scratch each time. For instance the symmetry of the addition is present three times:

Lemma plus_sym : (x,y:nat)(plus x y)=(plus y x).
Lemma Zplus_sym : (x,y:Z)(Zplus x y)=(Zplus y x).
Axiom Rplus_sym : (x,y:R)(Rplus x y)=(Rplus y x).

In this approach, the multiplicity continues for every new type that is introduced: complex numbers, polynomials, matrices, functions, etc.

5.2 The algebraic structure as a parameter

The naturals, integers and reals are all commutative groups under addition, and one can develop the theory of addition uniformly for groups. This group addition is polymorphic in (i.e. parameterized by) the particular group:

Gplus : (G:Group)(crr G)->(crr G)->(crr G).

where the carrier function crr:Group->Set gives the underlying set of a group.

For any concrete structure we wish to define, (e.g. the naturals) we must still define addition and prove it satisfies the commutative group axioms, but two significant advantages are gained. First, the theory of commutative groups need only be developed once, and will be inherited by each particular group. We can also declare abstract groups, which immediately inherit all the properties of groups. Second, and very important in large scale formalization, there are uniform names for all the properties of abelian groups. Users of the formalization need only consider one symmetry law:

Lemma Gplus_sym :
(G:Group; x,y:(crr G))(Gplus G x y)=(Gplus G y x).

This law is applicable to all groups.

5.3 Addition in the algebraic hierarchy

In the previous subsection, we considered the advantages of classifying structures as groups. Two refinements are necessary.

- In our framework there is not just one type of structure, but a hierarchy of structures: CSemi grp, CMonoid, CGroup, CRing, .... Addition is declared at the level of semi-groups, and inherited at more highly specified levels.
- Addition is not just an intensional function, but a setoid function.

\footnote{In fact, for no particular reason, symmetry of addition is declared at the level of monoids.}
Each structure is a ‘subtype’ of each simpler structure by a chain of forgetful coercions

```
cm_crr : CMonoid->Csemi_grp.
csg_crr : Csemi_group->CSetoid.
cs_crr : CSetoid->Set.
```

applied in the proper order.

The addition function, called csg_op, is one of the fields of the Csemi_grp record. It has type:

```
csg_op : (G:Csemi_group)
   (CSetoid_bin_fun (csg_crr G) (csg_crr G) (csg_crr G)).
```

This returns a CSetoid_bin_fun which is the type of a binary setoid function. From it one can retrieve the underlying type theoretic function by applying csbf_fun:

```
csbf_fun : (S1,S2,S3:CSetoid)(CSetoid_bin_fun S1 S2 S3)->
  (cs_crr S1)->(cs_crr S2)->(cs_crr S3).
```

Putting this together the sum of x and y in a semi-group G will be:

```
(csbf_fun (csg_crr G) (csg_crr G) (csg_crr G) (csg_op G) x y)
```

The syntax of Coq is powerful enough to allow this to be abbreviated by:

```
x[+]y
```

The G argument will be determined from the types of x and y. See Sect. 7 for discussion of the complexity of the underlying representation.

The symmetry of addition is declared by a field in the axioms of CMonoid. The form in which this property is used is a lemma with some definitions unfolded.

```
Lemma cm_commutes_unfolded: (M:CMonoid; x,y:M)(x[+]y [=]y[+]x).
```

Because of coercions, this lemma will work for every structure that can be coerced to a monoid. Those are the groups, rings, fields, the real and complex numbers, the polynomials, etc. The two advantages mentioned in the previous subsection hold across the whole algebraic hierarchy.

There are some technical restrictions on coercion, necessary to maintain the meaning of the implicit notations. For example, it is not possible to declare that the additive and multiplicative monoids of a ring both coerce to the same type of structure.

### 6 Partial functions and subsetoids

One of the main problems in formal mathematics is how to deal with partial functions. The type theoretic way to treat this problem is to add proof objects
as arguments to the functions; this is the approach that we followed in our framework.

The prototypical partial function is division. In the algebraic hierarchy the expression representing \( x/y \) will have three arguments, and be written

\[
x[/]y[/]H
\]

where \( H : (y[#]\text{Zero}) \) is a proof that \( y \) is apart from zero. This is actually managed by having a subisetoid of non-zero elements. Informally, the division function might be written:

\[
\div : F \times F_{\neq 0} \to F.
\]

Formally we define a type corresponding to \( F_{\neq 0} \) using the notion of a subisetoid.

The elements of a subisetoid_crr, the carrier of a subisetoid, are pairs of an element of setoid \( S \), and a proof that the element satisfies property \( P \).

Record subisetoid_crr [S:CSetoid; P:S->Prop]: Set :=
{ scs_elem : S;
  scs_prf : (P scs_elem)
}.

An instance of subisetoid_crr can be turned into a setoid in a canonical way, by inheriting the apartness and equality of \( S \), and showing that they satisfy the required properties. This is done via the map Build_SubCSetoid, which takes a setoid \( S \) and a predicate \( P \) over it, and returns the setoid of elements of \( S \) that satisfy \( P \).

Using this subisetoid construction we can define the setoid of the non-zeroes of a ring, together with functions nzinj and nzpro (injection and projection) that relate it to the original setoid. (We only give the types of the latter two functions.)

Variable F : CRing.
Definition NonZeros : CSetoid :=
  (Build_SubCSetoid F NonZeroP).
Definition nzinj : NonZeros->F := ...
Definition nzpro : (x:F)(x[#]Zero)->NonZeros := ...

Division in our framework is defined from the reciprocal function. This is a setoid function on the subisetoid of the non-zeroes:

cf_rcpcl : (CSetoid_un_op (NonZeros F))

Division therefore has type

cf_div : (F:Field)F->(NonZeros F)->F

and the expression \( x[/]y[/]H \) (parsed as \( x[/](y[/]H) \)) is shorthand for

\[
(cf\_div F x (nzpro F y H))
\]
The expression (\texttt{nzpro F y H}) represents \( y \) considered as an element of \( F_{\neq 0} \).

The proof terms that occur in expressions cause some calculation rules to have more conditions than one might expect. For instance the lemma formalizing:

\[
\frac{x}{y} \div z = \frac{x}{y} \cdot \frac{1}{z}
\]

is:

\[
\text{Lemma \ div \_div : } (x, y, z : F) (\texttt{nz}\_\texttt{y} y \# \texttt{Zero} ) (\texttt{nz}\_\texttt{z} z \# \texttt{Zero}) \\
(\texttt{nz}\_\texttt{y} y \# \texttt{z} \# \texttt{Zero}) \\
( (x / y \# y) [z / z] \cdot (y \# z)) \\
( (x / y) (y \# z) (y \# z)).
\]

In this lemma the condition \((y \# z) \# \texttt{Zero}\) is superfluous, as it is implied by \(y \# \texttt{Zero}\) and \(z \# \texttt{Zero}\). However, if we omit it (and plug in a proof using \texttt{nz}\_\texttt{y} and \texttt{nz}\_\texttt{z} in the place of \texttt{nz}\_\texttt{y}, \texttt{nz}\_\texttt{z})

\section{7 Syntax and complexity of terms}

We can’t use the customary operator symbols in our syntax because Coq doesn’t support overloading and the customary symbols are already in use. We indicate that we are using a setoid analogue of a normal operation by putting the operator in square brackets \([\text{ and }\]). So an equation like:

\[(x + y)^2 = x^2 + 2xy + y^2\]

becomes in our syntax:

\[(x \# y) [\cdot] (2) [\#] x [\cdot] (2) [\#] \texttt{Two} [\#] x [\#] y [\#] y [\cdot] (2)\]

and the equation:

\[p(X) = \sum_{i=0}^{n} a_i x^i\]

becomes:

\[p!x[#](\texttt{Sum} (0) \texttt{ n} [i : \texttt{nat}] (a i) [\#] x [\cdot] i)\]

Clearly our notation could be more readable.

However, the notational features of Coq provide significant benefits, and the official terms in our framework are much more complex than the notation suggests. For instance suppose we have \(x, y : \mathbb{R}\) (where \(\mathbb{R}\) is the type of the real numbers) then

\[x \# y\]

is an abbreviation of:

\[
\texttt{(csef\_fun (cse\_crr (cm\_crr (cg\_crr (cr\_crr (cf\_crr (cof\_crr \\
(\texttt{crl\_crr IR))))))))}
\]

\[
(\texttt{cse\_crr (cm\_crr (cg\_crr (cr\_crr (cf\_crr (cof\_crr (crl\_crr}
\]

Instead of what seems to be just one function symbol $[\cdot \cdot]$, the term actually contains 33 function symbols. This shows that the terms in our framework are relatively ‘heavy’. But note that most of this big term is inferable coercions. In Luo’s coercive subtyping [13] these parts of the term are actually elided, not just suppressed in printing. This may be an important optimization for large scale formal mathematics.

8 Conclusion

We presented a framework for writing algebraic expressions in the Coq proof assistant. The features of Coq that made our approach possible were:

- record types
- coercive subtyping
- implicit arguments

Similar features are also available in other systems (like for instance the Mizar system) and therefore something like our framework can be implemented in those systems.

In practice the framework that we have presented here works well. There is hardly any duplication of theory despite the great number of algebraic structures that we defined. For example, the theory of rings has been used for the rationals, the reals, the complex numbers and the polynomials. Apart from the reuse of theory, the reuse of notation (via a form of overloading introduced by the coercion mechanism) is also very convenient. Moreover this keeps user-level expressions reasonably concise.

8.1 Future work

There are various things that need to be investigated further:

**Better record types.** One would like the multiplicative monoid of a ring to be a coercive super-type of the ring type itself. Also, one would like a subsetoid to be a subtype of the setoid it is derived from. Both of these coercions don’t work well in the current version of Coq. Further research on how this situation can be improved is necessary. In [17] a start has been made with these investigations.
Structure of the hierarchy. The current hierarchy has been designed to make it possible to prove the fundamental theorem of algebra. This means that it is not as rich as one would like. For instance we don’t have non-commutative structures because they didn’t occur in our work.

One place where the hierarchy might be more refined is between CField, COrdField and CReals. Currently a number of the properties of fields of characteristic zero are derived in COrdField. This is not the right place because the complex numbers are not an ordered field so for it those results don’t apply. Similarly a number of the convergence notions are now defined only for CReals. However this is a subtype of COrdField so again these results don’t apply to the complex numbers. But the complex numbers do have a metric structure, and it would be desirable to have a type CMetricField for this.

Partial functions. The current way Coq deals with partiality is through proof terms in the expressions. This is unnatural. The PVS system [16] offers a different solution: in PVS a partial function is a total function on its domain. Whether a partial function is defined on an element is handled through so-called ‘type check conditions’, which may create extra proof obligations, but don’t show up in syntax. A similar approach is used in Nuprl, that has subset types. This approach works very well, but the price to be paid is that type checking becomes undecidable. This is not felt as a serious problem among PVS users. It is valuable to investigate whether a similar approach can be adapted to Coq.

Better syntax. The current syntax of our framework is not very readable. The integers and reals in the Coq standard library have custom parsers that allow for the more common algebraic notation. It would be valuable to build a parser like that for the algebraic hierarchy.

Classical logic. The current algebraic hierarchy is completely constructive. For many people it is irrelevant whether their reasoning is constructive or not. In their case classical logic would be much easier, and it would be useful to have a classical variant of the algebraic hierarchy. One could for instance define a notion of decidable setoid in which the equality is decidable. Combining this notion with the types of the algebraic hierarchy then would give a classical algebraic hierarchy.

8.2 Acknowledgements

Thanks to Dan Synek and the anonymous referees for valuable comments. Due to space limits we have not been able to incorporate all of them.

References

   URL: <http://www.cs.kun.nl/~freek/qed/qed.html>
   URL: <http://paulliac.inria.fr/jfla/2001/actes/04-delahaye.ps.gz>
   URL: <http://www.cs.kun.nl/~freek/mizar/mizarmanual.ps.gz>
On a Solution of Mutilated Checkerboard
Problem Using the Theorema Set Theory Prover

Wolfgang Windsteiger

No Institute Given

This paper was sent in only in PostScript, and is included as a separate file on the CD. In the printed copy this page is to be replaced by the printed form of the paper.
An Agent-oriented Approach to Reasoning

Christoph Benzmüller¹, Mateja Jamnik², Manfred Kerber² and Volker Sorge¹

¹Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany
{chris|sorge}@ags.uni-sb.de
http://www.ags.uni-sb.de/
²School of Computer Science, The University of Birmingham
Birmingham B15 2TT, England, UK
{M.Jamnik|M.Kerber}@cs.bham.ac.uk
http://www.cs.bham.ac.uk/

Abstract This paper discusses experiments with an agent oriented approach to automated and interactive reasoning. The approach combines ideas from two subfields of AI (theorem proving/proof planning and multi-agent systems) and makes use of state of the art distribution techniques to decentralise and spread its reasoning agents over the internet. It particularly supports cooperative proofs between reasoning systems which are strong in different application areas, e.g., higher-order and first-order theorem provers and computer algebra systems.

1 Introduction

The last decade has seen a development of various reasoning systems which are specialised in specific problem domains. Theorem proving contests, such as the annual CASC¹ competition, have shown that these systems typically perform well in their particular niches but often do poorly in others, or are not even applicable outside their specific niche. Whereas many hard-wired integrations of reasoning systems have been shown to be fruitful, rather few architectures have been discussed so far that try to extend the application range of reasoning systems by a flexible integration of a variety of specialist systems.

This paper discusses the implementation of experiments with an agent oriented reasoning approach, which has been presented as a first idea in [BJKS99]. The system combines different reasoning components such as specialised higher-order and first-order theorem provers, model generators, and computer algebra systems. It employs a classical natural deduction calculus in the background to bridge gaps between sub-proofs of the single components as well as to guarantee correctness of constructed proofs. The long term goal is to widen the range of mechanisable mathematics by allowing a flexible cooperation between specialist systems. This seems to be best achieved by an agent-based approach for a number of reasons. Firstly, from a software engineering point of view it offers a flexible way to integrate systems. Secondly, and more importantly, the agent-oriented

¹CADE ATP System Competitions, see also http://www.cs.jcu.edu.au/~tptp/.
approach enables a flexible proof search. This means that each single system – in form of an agent – can focus on parts of the problem it is good at, without the need to specify a priori a hierarchy of calls. However, we still focus on the construction of a single proof object and employ a centralised blackboard structure for communication. Therefore our agents mainly act as knowledge sources to blackboards, and our notion of agenthood is thus rather weak (compared to definitions of this term in some of the literature [Wei99]). The agents of our system can be described as autonomous agents capable of negotiation. However, extensive communication amongst the agents is currently also a weakness of our system, since too many resources are spent on communication. Hence, a future goal is to subsequently reduce this overhead by extending the agents’ reasoning capabilities and also by decentralising the approach. A discussion of particular agenthood aspects of our agents will be given in Section 4.

Using the agent paradigm enables us to overcome many limitations of static and hard-wired integrations. Furthermore, the agent based framework helps us to decontextualise and distribute conceptually independent reasoning processes as much as possible. An advantage over hard-wired integrations or even re- implementations of specialised reasoners is that it makes the reuse of existing systems possible (even without the need for a local installation of these systems). Accessing external systems is orchestrated by packages like MATHWEB [FJH+99] or the logic broker architecture [AZ01]. From the perspective of these infrastructure packages our work can be seen as an attempt to make strong use of their system distribution features.

Related agent-based theorem proving systems like [DF99, DD98, FI98, Wol98, Fis97] have demonstrated the feasibility and usefulness of agent oriented theorem proving. Our approach differs from previous work in various aspects. The main difference is that we are interested in combining heterogeneous systems, while we want to maintain a uniform representation of the overall proof attempt in natural deduction style. The proof results are explicitly represented in the core system in a higher-order natural deduction calculus. The core system is built on top of the proof development environment OMEGA [ea97] and its logic is a (sorted) higher-order logic based on Church’s simply typed λ-calculus. If full natural deduction translation packages for integrated external reasoners are available, then the proof contributions of these systems can be verified in the core system. In case they are not available, then we choose ad hoc representations and currently trust that these systems are correct. However, in contrast to several other researchers we believe that trust is not enough – in the long run we aim to add further translation mappings for external reasoners which will translate their contributions into the natural deduction calculus. Translating or representing external results in a uniform core system has also advantages with respect to the human-computer interface. Instead of dealing with several representation formalisms our approach enables the user to analyse and select contributions of external reasoners in the natural deduction calculus of the core system.
Our system currently uses about one hundred agents. They are split in several agent societies where each society is associated with one natural deduction rule/tactic of the base calculus. This agent set is extended by further agents encapsulating external reasoners. The encapsulation may be a direct one in case of locally installed external systems, or an indirect one via the MATHWEB framework, which facilitates their distribution over the internet. Employing numerous agents, amongst them powerful theorem provers which are computationally expensive, requires sufficient computation resources. Hence, it is crucial to build the whole system in a customisable and resource adaptive way. The former is achieved by providing a declarative agent specification language and mechanisms supporting the definition, addition, or deletion of reasoning agents (as well as some other proof search critical components and heuristics) even at run-time. For the latter, the agents in our framework can monitor their own performance, can adapt their capabilities, and can communicate to the rest of the system their corresponding resource information. This enables explicit (albeit currently still rudimentary) resource reasoning, facilitated by a specialised resource agent, and provides the basic structures for resource adaptive theorem proving.

The rest of the paper is structured as follows: Section 2 presents the main components of the system architecture. Experiments with the architecture are sketched in Section 3. In Section 4 we provide an overview of the features of our approach and discuss related work. A conclusion/outlook is given in Section 5.

2 System Architecture

The system architecture of our system is depicted in Fig. 1. The core of the system is written in Allegro Common Lisp and employs its multi-processing facilities. The choice of Common Lisp is due the fact that OMEGA, our base system, is implemented in this programming language; conceptually it can be implemented in any multi-processing framework.

Initial problems, partial proofs as well as completed proofs are represented in the Proof Data Structure [CS00] and the natural deduction infrastructure provided by the core system, OMEGA [ea97].

Our approach builds on the Reactive Suggestion Mechanism OANTS [BS01] as a reactive, resource adaptive basis layer of our framework. Triggered by changes in the proof data structure this mechanism dynamically computes applicable commands with their particular parameter instantiations and calls external reasoners into the current proof state. An important aspect is that all agent computations in this mechanism are de-sequentialised and distributed. The idea of this reactive layer is to receive results of inexpensive computations (e.g., the applicability of natural deduction rules) quickly while external reasoners search for their respective proof steps within the limits of their available resources, until a suggested command is selected and executed. A special resource agent receives performance data from the agents, which monitor their own performance, in order to adjust the system at run time. Heuristic criteria are used to dynamically filter and sort the list of generated suggestions. They are
then passed to the selector and/or the user. We give here some sensible heuristic criteria. Does a suggestion close a subgoal? Is a subgoal reduced to an essentially simpler context (e.g., reduction of higher-order problems to first-order or propositional logic)? Does a suggestion represent a big step in the search tree (proof tactics/methods) or a small step (base calculus rules)? Is the suggestion goal directed? How many new subgoals are introduced?

Agents as well as heuristic criteria can be added/deleted/modified at run time. Due to lack of space OANTS cannot be described here in detail; for this we refer the reader to [BS01].

OANTS provides agents for the basic natural deduction calculus computations. It also provides agents invoking additional proof tactics/methods and external reasoning systems. The latter are called indirectly via the MATHWEB system. We have extended the approach from [BS01] in the context of our work to be able to integrate partial proofs as results from the external reasoning systems into the overall proof as well as to store different alternative subproofs simultaneously. Moreover, we extended OMEGA's graphical user interface LOUI to be able to display different subproofs of external reasoners as choices for the user.

The Mathweb system realises calls to external reasoners which may be distributed over the internet. In our most recent experiments we extensively tested the new ONE-MATHWEB system which is based on a multi-broker architecture. Each broker has knowledge about its directly accessible reasoning systems, and also about urls to other ONE-MATHWEB brokers on the internet. For example, in our experiments we gained access to the computer algebra system MAPLE running in Saarbrücken simply by informing the Birmingham MATHWEB broker (which for license reasons cannot locally offer a MAPLE service) about the existence and url of the Saarbrücken broker. The Saarbrücken broker then connects

Figure 1: System architecture.
the Birmingham broker indirectly with the MAPLE service. Currently our system links up with the computer algebra systems MAPLE and GAP running in Saarbrücken, and locally with the higher-order theorem provers LEO and TPS, the first-order theorem prover OTTER (employed also as our propositional logic specialist), and SATCمو (employed as a model generator). MATHWEB is described in detail in [FHJ+99].

Once the reactive suggestion mechanism dynamically updates and heuristically sorts the list of suggestions, which are commands together with their particular parameter instantiations, it passes the list on to the selector. Its main task is to automatically execute the heuristically preferred command, and hence, initiate an update of the proof data structure. Furthermore, the selector stores the non-optimal, alternative command suggestions in a special store. The information in this store is used when backtracking to a previous state in the proof data structure becomes necessary. Instead of a complete initialisation the reactive suggestion mechanism is then simply initialised with the already computed backtracking information for the current proof context. Backtracking is caused when the reactive layer produces no suggestions or when a user defined maximal depth\footnote{Iterative deepening proof search wrt. to the maximal depth is conceptually feasible but not realised yet.} in the proof data structure is reached.

The backtrack store maintains backtracking information for the proof data structure. This information includes representations of the suggestion computations that have been previously computed but not executed. Additionally the store maintains the results of external system calls modulo their translation in the core natural deduction calculus. That is, the immediate translation of external system results is also done by the reactive suggestion layer, and the results of these computations are memorised for backtracking purposes as well. If the system or the user selects to apply the result of an external system, the proof data structure is updated with the translated proof object. Future work will include investigating whether the backtrack store should be merged with the proof data structure. The idea is that each single node in a proof directly maintains its backtracking alternatives instead of using an indirect maintenance via the backtrack store.

The tasks of the user interface in our framework are:

1. To visualise the current proof data structure and to ease interactive proof construction. For this purpose we employ OMEGA’s graphical user interface LOUI [SHB+99].
2. To dynamically present to the user the set of suggestions, which pop up from the reactive layer to the user, and to provide support for analysing or executing them. This is realised by structured and dynamically updated pop-up windows in LOUI.
3. To provide graphical support for analysing the results of external systems, that is, to display their results after translation/representation in the proof data structure. We achieve this by extending LOUI so that it can switch
between the global proof data structure and locally offered results by external systems.

4. To support the user in interacting with the automated mechanism and in customising agent societies at run-time.

From an abstract perspective, our system realises proof construction by going through a cycle which consists of assessing the state of the proof search process, evaluating the progress, choosing a promising direction for further search and redistributing the available resources accordingly. If the current search direction becomes increasingly less promising then backtracking to previous points in the search space is possible. Only successful or promising proof attempts are allowed to continue searching for a proof. This process is repeated until a proof is found, or some other terminating condition is reached.

3 Experiments

In this section we report on experiments we conducted with our system to demonstrate the usefulness of a flexible combination of different specialised reasoning systems. Among others we examined different problem classes:

1. Set examples which demonstrate a cooperation between higher-order and first-order theorem provers. For instance, prove:
\[\forall x, y, z. (x = y \cup z) \iff (y \subseteq x \land z \subseteq x \land \forall u (y \subseteq v \land z \subseteq v) \Rightarrow (x \subseteq v)\]

2. Set equations whose validity/invalidity is decided in an interplay of a natural deduction calculus with a propositional logic theorem prover and model generator. For instance, prove or refute:
   (a) \(\forall x, y, z. (x \cup y) \cap z = (x \cap z) \cup (y \cap z)\)
   (b) \(\forall x, y, z. (x \cup y) \cap z = (x \cup z) \cap (y \cup z)\)

3. Concrete examples about sets over naturals where a cooperation with a computer algebra system is required. For instance (\(gcd\) and \(lcm\) stand for the ‘greatest common divisor’ and the ‘least common multiple’):
\[\{x | x > gcd(10, 8) \land x < lcm(10, 8)\} = \{x | x < 40\} \cap \{x | x > 2\}\]
This set is represented by the lambda expression
\[(\lambda x. x > gcd(10, 8) \land x < lcm(10, 8)) = (\lambda x. x < 40) \cap (\lambda x. x > 2)\]

4. Examples from group theory and algebra for which a goal directed natural deduction proof search is employed in cooperation with higher order and first order specialists to prove equivalence and uniqueness statements. These are for instance of the form
\[\exists o \cdot \text{Group}(G, o) \iff [\exists * \cdot \text{Monoid}(M, *) \land \text{Inverses}(M, *, \text{Unit}(M, *))]\]
Here \(\text{Group}\) and \(\text{Monoid}\) refers to a definition of a group and a monoid, respectively. \(\text{Inverses}(M, *, \text{Unit}(M, *))\) is a predicate stating that every element of \(M\) has an inverse element with respect to the operation \(*\) and the identity \(\text{Unit}(M, *)\). \(\text{Unit}(M, *)\) itself is a way to refer to that unique element of \(M\) that has the identity property.

We will sketch in the following how the problem classes are tackled in our system in general and how the proofs of the concrete examples work in particular.
3.1 Set examples

The first type of examples is motivated by the shortcomings of existing higher-order theorem provers in first-order reasoning. For our experiments we used the LEO system [BK98], a higher-order resolution prover, which specialises in extensionality reasoning and is particularly successful in reasoning about sets.

Initialised with a set problem LEO tries to apply extensionality reasoning in a goal directed way. On an initial set of higher-order clauses, it often quickly derives a corresponding set of essentially first-order clauses. Depending on the number of generated first-order and other higher-order clauses LEO may get stuck in its reasoning process, although the subset of first-order clauses could be easily refuted by a first-order specialist.

For our examples the cooperation between LEO and the first-order specialist OTTER works as depicted in Fig. 2. The initial problem representation in the proof data structure is described in Part 1 of Fig. 2. The initialisation triggers the agents of the reactive suggestion layer which start their computations in order to produce suggestions for the next proof step.

The agent working for LEO first checks if there is any information from the resource agent that indicates that LEO should stay passive. If not, it checks whether the goal $C$ is suitable for LEO by testing if it is a higher-order problem. In case the problem is higher-order the agent passes the initial problem consisting of the goal $C$ and the assumptions $P_1, \ldots, P_n$ to LEO. While working on the input problem (as indicated by the shaded oval in Part 2 of Fig. 2) LEO derives (among others) various essentially first-order clauses (e.g., $\forall x \exists y F_1 \land \ldots \land F_n$). For the particular type of cooperation described here, it is important that after a while this subset becomes large enough to be independently refutable. If after consuming all the resources made available by the reactive suggestion layer LEO still fails to deliver a completed proof, it then offers a partial proof consisting of a subset of first-order and essentially first-order clauses (after translation into prefix normal form, e.g., $\exists x \forall y F_1' \land \ldots \land F_n'$, where the $F_i'$ are disjunctions of

\footnote{By essentially first-order we mean a clause set that can be tackled by first-order methods. It may still contain higher-order variables, though.}
the literals of FO, and \( \pi \) stands for the sequence of all free variables in the scope). In case LEO's suggestion wins over the suggestions computed by other agents, its partial result is represented in the proof data structure and the reactive suggestion mechanism is immediately triggered again to compute a suggestion for the next possible proof step. Since LEO's partial result is now the new subgoal of the partial proof, first-order agents, like the one working for OTTER, can pick it up and ask OTTER to prove it (see Part 3 of Fig. 2). If OTTER signals a successful proof attempt before consuming all its given resources, its resolution proof is passed to the natural deduction translation module TRAMP [Mei00], which transforms it into a proper natural deduction proof on an assertion level.

We experimented with 121 simple examples, that is, examples that can be automatically proved by LEO alone. The results showed that the command execution interval chosen by the selector is crucial, since it determines the computation time \( ct \) made available to the external systems.

- If \( ct \) is sufficiently high, then the problem is automatically proved by LEO (in case of simple examples that can be solved by LEO alone).
- If \( ct \) is not sufficient for LEO to come up with a proof, but still enough to produce a refutable subset of essentially first-order clauses, then a cooperative proof is constructed as described above.
- If \( ct \) is not sufficient to even guarantee a subset of refutable essentially first-order clauses, then the problem is tackled purely on natural deduction level, however not necessarily successfully.

We also solved several examples which cannot be solved with LEO alone. One of them is the concrete example given above, which, to our knowledge, cannot be easily solved by a single automated theorem prover. In our experiments, LEO alone ran out of memory for the above problem formulation, and OTTER alone could not find a proof after running 24 hours in auto mode on a first-order formulation of the problem. Of course an appropriate reformulation of the problem can make it simple for systems like OTTER.

### 3.2 Set equations

The second type of set examples illustrates a cooperation between automated natural deduction agents, a propositional prover and a model generator. The proofs follow a well-known set theoretic proof principle: they are constructed first by application of simple natural deduction agents that reduce the set equations by applying set extensionality and definition expansion to a propositional logic statement. This statement is then picked up by an agent working for a propositional logic prover (here we use again OTTER encapsulated in another agent shell with a slightly modified applicability check and a different representation translation approach) and a counter-example agent which employs SATCHMO.

\[\text{\footnotesize N} \] While TRAMP already supports the transformation of various machine oriented first-order proof formats, further work will include its extension to higher-order logic, such that also the proof step justified in Fig. 2 with 'LEO-derivation' can be properly expanded into a verifiable natural deduction proof.
The logic statement is then either proved or refuted. Thus, valid and invalid statements are tackled analogously in all but the last step.

In case (2a) of our concrete examples several \( V_T \) (universal quantification introduction in backward reasoning) applications introduce \((a \cup b) \cap c = (a \cap c) \cup (b \cap c)\) as new open subgoal. Set extensionality gives us \( \forall u. u \in (a \cup b) \cap c \Leftrightarrow u \in ((a \cap c) \cup (b \cap c)) \). A further \( V_T \) application and subsequent definition expansions (where \( a \cup b := \lambda z. (z \in a) \lor (z \in b) \), \( a \cap b := \lambda z. (z \in a) \land (z \in b) \), and \( u \in a := a(u) \)) reduce this goal finally to \( (a(d) \lor b(d)) \land c(d) = (a(d) \land c(d)) \lor (b(d) \land c(d)) \) which contains no variables and which is a trivial task for any propositional logic prover. In case (2b) we analogously derive \( (a(d) \lor b(d)) \land c(d) = (a(d) \lor c(d)) \land (b(d) \lor c(d)) \), but instead of employing the propositional prover, the system now uses a model generator which presents the counter-model \( a(d), b(d), \neg c(d) \). That is, it points to the set of all \( d \) such that \( d \in a, d \in b \), but \( d \notin c \). Hence, the model generator comes up with a counter-example to the expression in (2b).

Future work includes integrating facilities for the visualisation of these counterexamples by Venn diagrams in LOUI.

We have experimented with an automatically and systematically generated testbed consisting of possible set equations involving \( \cap, \cup \), set-minus operations up to nesting depth of 5 in maximally 5 variables. We classified 10000 examples with our system discovering 988 correct and 9012 false statements. Naturally, the correct statements are probably also solvable with the cooperation of LEO and OTTER.

### 3.3 Examples with computer algebra

The next type of examples has cross-domain character and requires a combination of domain specific systems. In order to tackle them we added a simplification agent which links the computer algebra system MAPLE to our core system. As an application condition this agent checks whether the current subgoal contains certain simplifiable expressions. If so, then it simplifies the subgoal by sending the simplifiable subterms (e.g., \( x > \gcd(10, 8) \)) via MATHWEB to MAPLE and replaces them with the corresponding simplified terms (e.g., \( x > 40 \)). Hence, the new subgoal suggested by the simplification agent is: \((\lambda x. x > 2) \land x < 40\) = \((\lambda x. x < 40) \land (\lambda x. x > 2)\). Since no other agent comes up with a better alternative, this suggestion is immediately selected and executed. Subsequently, the LEO agent successfully attacks the new goal after expanding the definition of \( \cap \). We have successfully solved 50 problems of the given type and intend to generate a large testbed next.

### 3.4 Group theory and algebra examples

The group theory and algebra examples we examined are rather easy from a mathematical viewpoint, however, can become non-trivial when painstakingly formalised. An example are proofs in which particular elements of one mathematical structure have to be identified by their properties and transferred to
their appropriate counterparts in an enriched structure. The equivalence statement given above where the unit element of the monoid has to be identified with the appropriate element of the group are in this category. In higher-order this can be done most elegantly using the description operator \( \iota \) (cf. [And72] for description in higher-order logics) by assigning to the element in the group the unique element in the monoid that has exactly the same properties. In the context of our examples we employed description to encode concepts like the (unique) unit element of a group by a single term that locally embodies the particular properties of the encoded concept itself. If properties of the unit element are required in a proof then the description operator has to be unfolded (by applying a tactic in the system) and a uniqueness subproof has to carried out. However, an open problem is to avoid unnecessary unfoldings of the description operator as this may overwhelm the proof context with unneeded information.

The idea of the proofs is to divide the problems into smaller chunks that can be solved by automated theorems provers and if necessary to deal with formulae involving description. The ND search procedure implemented in OANTS has the task to successively simplify the given formulae by expanding definitions and applying ND inferences. After each proof step the provers try to solve the introduced subproblems. If they all fail within the given time bound the system proceeds with the alternative ND inferences. The quantifier rules introduce Skolem variables and functions when eliminating quantifications. These are constrained either by the application of a generalised Weaken rule, using higher-order unification, or by the successful solution of subproblems by one of the provers, which gives us the necessary instantiation. Problems involving higher-order variables (for which real higher-order instantiations are required) can generally not be solved (in this representation) by first-order provers. However, once an appropriate instantiation for the variables has been computed a first-order prover can be applied to solve the remaining subproblems. Substitutions for introduced Skolem variables are added only as constraints to the proof, which can be backtracked if necessary.

When a point is reached during the proof where neither applicable rules nor solutions from the provers are available, but the description operator still occurs in the considered problem, two theorems are applied to eliminate description. This results in generally very large formulae, which can then again be tackled with the ND rules and the theorem provers.

In our experiments with algebra problems we have successfully solved 20 examples of the described type.

Our experiments show that the cooperation between different kinds of reasoning systems can fruitfully combine their different strengths and even out their respective weaknesses. In particular, we were able to successfully employ LEO's extensionality reasoning with Otter's strength in refuting large sets of first-order clauses. Likewise, our distributed architecture enables us to exploit the computational strength of Maple in our examples remotely over the internet. As particularly demonstrated by the latter example class the strengths of exter-
nal systems can be sensibly combined with domain specific tactics and methods, and natural deduction proof search.

Note that our approach does not only allow the combination of heterogeneous systems to prove a problem, but it also enables the use of systems with opposing goals in the same framework. In our examples the theorem prover and the model generator work in parallel to decide the validity of the current goal.

Although many of our examples deal with problems in set theory they already show that the cooperation of differently specialised reasoning systems enhances the strengths of automated reasoning. The results also encourage the application of our system to other areas in mathematics in the future. However, there is a bottleneck for obtaining large proofs, namely the translation between the different systems involved, in particular, in the presence of large clause sets.

4 Discussion

Our work is related to blackboard and multi-agent systems in general, and to approaches to distributed proof search and agent-oriented theorem proving in particular. Consequently, the list of related work is rather long and we can mention only some of it. We first summarise different facets of our approach which we then use to clarify the differences to other approaches and to motivate our system design objectives. Our system:

1. aims to provide a cognitively adequate assistant tool to interactively and/or automatically develop mathematical proofs;
2. supports interaction and automation simultaneously and integrates reactive and deliberative proof search;
3. maintains a global proof object in an expressive higher order language in which results of external systems can be represented;
4. employs tools as LOUI [SHB+99] or P.rex [Fie01a, Fie01b] to visualise and verbalise proofs, i.e., communicate them on a human oriented representation layer;
5. couples heterogeneous external systems with domain specific tactics and methods and natural deduction proof search; i.e., our notion of heterogeneity comprises machine oriented theorem proving as well as tactical theorem proving/proof planning, model generation, and symbolic computation;
6. reuses existing reasoning systems and distributes them via MATHWEB (In order to add a new system provided by MATHWEB the user has to: a) provide an abstract inference step/command modelling a call to the external reasoner, b) define the parameter agents working for it, and c) (optional) adapt the heuristic criteria employed by the system to rank suggestions. Due to the declarative agent and heuristics specification framework these steps can be performed at run time.);
7. supports competition (e.g., proof versus countermodel search) as well as cooperation (e.g., exchange of partial results);
(8) follows a sceptical approach and generally assumes that results of external reasoning system are translated in the central proof object (by employing transformation tools such as Tramp [Mei00]) where they can be proof-checked;

(9) employs resource management techniques for guidance;

(10) supports user adaptation by enabling users to specify/modify their own configurations of reasoning agents at run-time, and to add new domain specific tactics and methods when examining new mathematical problem domains;

(11) stores interesting suboptimal suggestions in a backtracking stack and supports backtracking to previously dismissed search directions;

(12) supports parallelisation of reasoning processes on different layers: term-level parallelisation is achieved by various parameter agents of the commands/abstract inferences, inference-level parallelisation is supported by the ability to define new powerful abstract inferences which replace several low level inferences by a single step (a feature inherited from the integrated tactical theorem proving paradigm), and proof-search-level parallelisation is supported by the competing reasoning systems.

Taken individually none of the above ideas is completely new and for each of these aspects exists related work in the literature. However, it is the combination of the above ideas that makes our project unique and ambitious.

A taxonomy of parallel and distributed (first-order) theorem proving systems is given in [Bon99, Bon01]. As stated in (12), our approach addresses all three classification criteria introduced there: parallelisation on term, inference, and search level. However, full or-parallelisation is not addressed in our approach yet. This will be future work.

A very related system is the Techs approach [DF99] which realises a cooperation between a set of heterogeneous first-order theorem provers. Partial results in this approach are exchanged between the different theorem provers in form of clauses, and different referees filter the communication at the sender and receiver side. This system clearly demonstrates that the capabilities of the joint system are bigger than those of the individual systems. Techs' notion of heterogeneous systems, cf. (5) above, however, is restricted to a first-order context only. Also symbolic computation is not addressed. Techs [DF99] and its even less heterogeneous predecessors Teamwork [DK96] and Discount [ADF95] are much more machine oriented and less ambitious in the sense of aspects (1)–(4). However, the degree of exchanged information (single clauses) in all these approaches is higher than in our centralised approach. Unlike in the above mentioned systems, our interest in cooperation, however, is in the first place not at clause level, but on subproblem level, where the subproblem structure is maintained by the central natural deduction proof object. Future work includes investigating to what extent our approach can be decentralised, for instance, in the sense of Techs, while preserving a central global proof object.

In contrast to many other approaches we are interested in a fully sceptical approach, cf. (8) and the results of some external reasoners (e.g., for Otter Tps, and partially for computer algebra systems) can already be expanded and
proof checked by translation in the core natural deduction calculus. However, for some external systems (e.g., LEO) the respective transformation tools still have to be provided. While they are missing, the results of these systems, modelled as abstract inferences in natural deduction style, cannot be expanded.

Interaction and automation are addressed by the combination of ILF & TECS [DD98]. With respect to aspects (6)–(12), especially (10), there are various essential differences in our approach. The design objectives of our system are strongly influenced by the idea to maintain a central proof object which is manipulated by the cooperating and competing reasoning agents, and mirrors the proof progress. This central natural deduction proof object especially eases user interaction on a human oriented layer, cf. (3) and (4), and supports scepticism as described above. In some sense, external systems are modelled as new proof tactics. Extending the background calculus and communication between them is currently only supported via the system of blackboards associated with the current focus of the central proof object. This relieves us from addressing logical issues in the combination of reasoning systems at the proof search layer. They are subordinated and only come into play when establishing the soundness of contributions of external reasoners by expanding their results on natural deduction layer. A centralised approach has advantages in the sense that it keeps the integration of heterogeneous systems, with possibly different logical contexts, simple and it only requires n different proof (or result) transformation tools to natural deduction arguments. In particular the overall proof construction is controlled purely at the natural deduction layer.

However, experiments indicated that aside from these advantages, the bottleneck of the system currently is the inefficiency in the cooperation of some external systems, especially of homogeneous systems specialises in resolution style proving which cannot directly communicate with each other. Future work therefore includes investigating whether the approach can be further decentralised without giving up much of the simplicity and transparency of the current centralised approach.

With the centralisation idea, we adopted a blackboard architecture and our reasoning agents are knowledge sources of it. In the terminology of [Wei99] our reasoning agents can be classified as reactive, autonomous, pro-active, cooperative and competitive, resource adapted, and distributed entities. They, for instance, still lack fully deliberative planning layers and social abilities such as means of explicit negotiation (e.g., agent societies are defined by the user in OANTS and, as yet, not formed dynamically at run-time [BS91]). In this sense, they are more closely related to the HASP [NFAR82] or POLIGON [Ric89] knowledge sources than to advanced layered agent architectures like INTERRAP [Müller97]. However, in future developments a more decentralised proof search will make it necessary to extend the agenthood aspects in order to enable agents to dynamically form clusters for cooperation and to negotiate about efficient communication languages.
5 Conclusion

In this paper we presented an approach to agent-based reasoning. Our framework is based on concurrent suggestion agents working for natural deduction rules, tactics, methods, and specialised external reasoning systems. The suggestions by the agents are evaluated after they are translated into a uniform data representation, and the most promising direction is chosen for execution. The alternatives are stored for backtracking. The system supports customisation and resource adapted and adaptive proof search behaviour.

The main motivation is to develop a powerful system for tackling, for instance, cross domain examples, which require a combination of reasoning techniques with strengths in individual domains. However, our motivation is not to outperform specialised systems in their particular niches. The agent paradigm was chosen to enable a more flexible integration approach, and to overcome some of the limitations of hardwired integrations (for instance, the brittleness of traditional proof planning where external systems are typically called within the body of proof methods and typically do not cooperate very flexibly).

A cognitive motivation for a flexible integration framework presented in this paper is given from the perspective of mathematics and engineering. Depending on the specific nature of a challenging problem, different specialists may have to cooperate and bring in their expertise to fruitfully tackle a problem. Even a single mathematician possesses a large repertoire of often very specialised reasoning and problem solving techniques. But instead of applying them in a fixed structure, a mathematician uses own experience and intuition to flexibly combine them in an appropriate way.

The experience of the project points to different lines of future research. Firstly, the agent approach offers an interesting framework for combining automated and interactive theorem proving on a user-oriented representation level (and in this sense it differs a lot from the mainly machine-oriented related work). This approach can be further improved by developing a more distributed view of proof construction and a dynamic configuration of cooperating agents. Secondly, in order to concurrently follow different lines of search (or-parallelism), a more sophisticated resource handling should be added to the system. Thirdly, the communication overhead for obtaining large proofs is the main performance bottleneck. More efficient communication facilities between the different systems involved have to be developed. Contrasting the idea of having filters as suggested in [DF99] we also want to investigate whether in our context (expressive higher-order language) abstraction techniques can be employed to compress the exchanged information (humans do not exchange clauses) during the construction of proofs.

Further future work includes improving several technical aspects of the current OMEGA environment and the prototype implementation of our system that have been uncovered during our experiments. We would also like to test the system in a real multi-processor environment, where all agents for external reasoners can be physically rather than indirectly (via MATHWEB) distributed. Furthermore, we will integrate additional systems and provide further representation
translation packages. The agents' self-monitoring and self-evaluation criteria, and the system's resource adjustment capabilities will be improved in the future. We would also like to employ counter-example agents as indicators for early backtracking. Finally, we need to examine whether our system could benefit from a dynamic agent grouping approach as described in [FW95], or from an integration of proof critics as discussed in [IB95].

Acknowledgements

For advice, support, and encouragement we thank Alan Bundy, Michael Fisher, Malte Hübner, Andreas Franke, and Jürgen Zimmer. This work was supported by EPSRC grants GR/M99644 and GR/M22031 (Birmingham), and the SFB 378 (Saarbrücken).

References


[AZ01] A. Armando and D. Zini. Interfacing computer algebra and deduction systems via the logic broker architecture. In [CAL01].


[BS01] C. Benzmüller and V. Sorge. OANTS – An open approach at combining Interactive and Automated Theorem Proving. In [CAL01].


[Fie01a] A. Fiedler. Dialog-driven adaptation of explanations of proofs. Accepted for LICAL-01.

[Fie01b] A. Fiedler. P.rex: An interactive proof explainer. Accepted for IJCAR-01.


Towards Mathematical Agents – Combining MATHWEB-SB and the LBA

Jürgen Zimmer\textsuperscript{1,2}, Alessandro Armando\textsuperscript{2}, and Corrado Giomi\textsuperscript{2}

\textsuperscript{1} AGS, Fachbereich 14, Universität des Saarlandes, Saarbrücken, Germany
jzimmer@mathweb.org
\textsuperscript{2} MRG, DIST, Università di Genova, Italy
{armando,nembo}@mrg.dist.unige.it

Abstract This paper describes ongoing research aimed at interconnecting two emerging architectures for distributed mathematical problem solving: the MATHWEB Software Bus (MATHWEB-SB) and the Logic Broker Architecture (LBA). The bridge between MATHWEB-SB and LBA is achieved by a common interface based on KQML and OPEN-MATH. We augmented both architectures with MATHWEB agents that communicate via a common interface sending KQML messages. The proposed interface offers an appropriate abstraction for combining architectures like the MATHWEB-SB and the LBA and allows external systems to easily access the mathematical services of both architectures. As a case study we show that MATHWEB agents encapsulating the RDL system, the mathematical knowledge base MBASE and the OMEGA system – sited in the LBA or in the MATHWEB-SB respectively – can communicate via our bridge and perform cooperative problem solving.

1 Introduction

Modern applications of automated reasoning techniques require open software systems that support modularization, inter-operability, robustness and scalability. The MATHWEB Software Bus (MATHWEB-SB) [FK99] and the Logic Broker Architecture (LBA) [AZ00] are two platforms that have been developed in recent years in the AGS\textsuperscript{1} and the MRG\textsuperscript{2} respectively to meet these requirements. Although, these architectures are rather similar, they both have some advantages over the other. For instance, the MATHWEB-SB already offers the means for building a robust and scalable system and it contains a load balancing mechanism. The LBA is, for instance, based on the industrial standard CORBA for inter-operability and offers abstract mathematical services.

In order to benefit from the merits of both, the MATHWEB-SB and the LBA, we have built a bridge between the two architectures that (i) preserves the original functionalities of the component architectures, (ii) enables the reasoning services integrated in one architecture to readily use to the services offered by the other, and (iii) allows external client applications to have uniform access to the mathematical services in both systems via a standardized interface.

* The author is supported by the European Union IHP grant HPRN-CT-2000-00102.
In order to realize such a combined system, we followed the ideas of Armando, Kohlhase, and Rannse [AKR00] who presented an interaction protocol for mathematical services based on the Knowledge Query and Manipulation Language (KQML) [LF94, Lab96] and the definition of Open Mechanized Reasoning Systems (OMRS) specification framework [GPT94].

"In order to achieve a joint system that uses much of the current functionality [...] it is sufficient to augment each of the systems above by an agent that does the KQML-communication and serves as a bridge."

We follow the Agent Oriented Programming paradigm (AOP) [Sho90] and present a preliminary structure of mathematical reasoning agents, which we call MathWeb agents. MathWeb agents communicate with the KQML with OpenMath [3] formulas as a content. We extended the reasoning services of the MathWeb-SB and the LBA to MathWeb agents and augmented both architectures by KQML facilitators which route KQML messages and communicate with the Hyper Text Transfer Protocol (HTTP). We show that the bridge preserves the original functionality of the two systems and that MathWeb agents, running in the MathWeb-SB and in the LBA, can communicate via the bridge and perform cooperative problem solving. Our implementation also allows external systems to easily access the mathematical services in both systems via a standardized interface.

In a case study we use the decision procedure for linear arithmetic which is available in RDL [ACR01] as an oracle to retrieve instantiations for meta-variables within the proof planner of the Omega system [ea97]. During the problem solving process both systems access the mathematical knowledge base MBASE [KF00].

The structure of the paper is as follows. In Section 2 we briefly introduce and compare the MathWeb-SB and the LBA. In section 3 we give a technical description of the bridge. Finally, we present our case study in section 4.

It is worth to be mentioned once again, that this paper presents ongoing research. Therefore, the reader should not expect final results.

2 The MathWeb Software Bus and the LBA

2.1 The MathWeb Software Bus

The MathWeb Software Bus (MathWeb-SB) [FK99] for distributed automated theorem proving supports the connection of a wide range of mathematical services by a common software bus. The MathWeb-SB provides the functionality to turn existing theorem proving systems, computer algebra systems, and miscellaneous tools into mathematical services that are homogeneously integrated into a proof development environment.

Using the MathWeb-SB technology we developed the MathWeb system which is a stable network of mathematical services. The MathWeb system is implemented in Mozart Oz [grob], a multi-paradigm object-oriented programming language which fully supports concurrent and distributed programming and allows to simply distribute applications over the Internet. The services of the current MathWeb system are used permanently by client applications, e.g. by the Omega system [ea97], DORIS [BBK99], and the ActiveMath system [MAlf01]. MathWeb currently integrates many different reasoning and computation systems, like for instance, automated theorem provers (e.g. Otter, sspp, etc.), computer algebra systems (CASs) (e.g. Maple, and GAP),
translation services, the mathematical knowledge base MBASE [KF00], and constraint solving systems (e.g. CoSIE [Zim00] and Chorus [KN00]).

Fig.1 shows part of the MathWeb system as it is currently running. In MathWeb, meta-services (MS) offer the mathematical services (e.g. an ATP, or a CAS) to their local MathWeb broker. MathWeb brokers register and unregister to each other and, therefore, build a dynamic web of brokers. Client applications, like the Omega system, DORIS or a CGI-script, connect to one of the MathWeb and request services. If the requested service is not offered by a local meta-service, the broker forwards the request to all other brokers until the service is found (accept) or it is not found anywhere in the MathWeb system (deny). If the requested service is found, the client application receives a reference to a newly created service object and can directly send messages to the object. MathWeb-SB currently offers three interfaces to connect to a broker, namely Mozart’s distributed programming interface, CGI-script access via an HTTP server, and access via an XMLRPC server.

2.2 The Logic Broker Architecture

In [AZ00] Armando and Zini have presented the Logic Broker Architecture (LBA), a framework which provides the needed infrastructure for making mechanised reasoning systems inter-operate. The LBA provides location transparency and a way to forward requests for logical services to appropriate reasoning systems via a simple registration/subscription mechanism. The basic means to achieve this are the CORBA [Groa] standard for client-server applications and the OPENMATH standard. CORBA is specifically designed to interface software systems and provides location transparency for free.

The basic schema of the LBA is depicted in Fig.2, where a client reasoning system C gets access to the services provided by a service server S via the Logic Broker (LB). A reasoning system S can register a logical service to the LB by sending the LB a message of the form register(lSs), where lSs is a specification of the logical service it
The current implementation of the LBA supports the use of a single central broker.

The reader may have already noted that the MathWeb-SB and the LBA have a similar structure. They both have the notion of one or more logical services. A mathematical application (server) exposes services through the service interface. Each composite service knows nothing about the lower-level communication details. Only the composite service is exposed. The MathWeb-SB offers some features that are not offered by the LBA. It offers a dynamic and web-based service for building a robust and scalable service. MathWeb-SB can be accessed by three different services: MathWeb-SB, MathWeb-DB, and MathWeb-LB. MathWeb-SB integrates various tools or system components which are essential to keep a stable system up and running.

Figure 2: The Logic Broker Architecture

is able and willing to provide (e.g., prove simplification, etc.). An application can subscribe to the LBA as a client by reserving the service name (service). As a result, if the LBA searches the database of registered services for a service name (service), which is a specification of the logical service, then the client searches the database of registered services for a service that has the same interface as the required one. The LBA searches the database of registered services for a service that has the same interface as the required one. The LBA then returns the service object (S-Matcher) to handle the query. The service object (S-Matcher) is then used to handle the query. The service object (S-Matcher) is then used to handle the query.

The current implementation of the LBA supports the use of a single central broker.
– On the other hand, the LBA has some advantages over the MATHWEB-SB: It follows the idea of abstract logical services (e.g. factor, simplify, etc.) where in the MATHWEB-SB, mathematical services always offer the full functionality of the underlying reasoning system (e.g. the eval function of the CAS MAPLE). Moreover, the LS Matcher of the LBA is going to ensure the logical soundness of system integration.

The MATHWEB-SB and the LBA are based on different implementation platforms. Therefore, to link the reasoning services provided by the two architectures, we have to translate service requests of one architecture to requests in the other. This translation must be sound, i.e. it must preserve the semantics of the request. The bridge proposed in the following section uses OPENMATH symbols from an Extra content dictionary (reasyms.cd) to describe mathematical services. We hope that in the future the OPENMATH society will come to a consensus about the semantics of symbols describing mathematical services in the same manner than it is done for the Core OPENMATH symbols.

3 Building the Bridge

In this section, we present the bridge which combines the MATHWEB-SB and the LBA. We extended the reasoning systems of both architectures to MATHWEB agents and developed a common interface which is independent of the underlying implementation platforms. The work presented here is part of the endeavor to apply agent oriented programming techniques to automated reasoning in order to construct a web of heterogeneous mathematical agents which perform distributed problem solving, handle shared proof objects, and dynamically coordinate their behavior given a problem at hand. The reader should regard our bridge as a first step towards this ultimate goal and as an attempt to gain first experience with KQML communication between MATHWEB agents.

In the following two section we first give a brief introduction into agent oriented programming and the KQML language. In section 3.2 we describe the structure of MATHWEB agents which are build according to the AOP paradigm and which communicate with KQML. Finally, we describe the details of our bridge in section 3.3.

3.1 Agent Oriented Programming

The term Agent Oriented Programming (AOP) was coined by Shoham in 1990 [Sho90]. It is a “new programming paradigm, based on a societal view of computation”. The key idea is that of directly programming software agents which encapsulate arbitrary, traditional software applications. These agent shells are able to interface and control the operation of the embedded services. The basic means for the interaction between agents is a common Agent Communication Language (ACL) which enable the agents to coordinate their behavior, i.e., steer the embedded applications by exchanging beliefs, goals, and intentions.

KQML [LF94, Lab96] is a communication language for software agents which supports the exchange of information about the (virtual) knowledge bases (VKB) of the agents. KQML is both a message format and a message-handling protocol to support

\footnote{The interested reader should wait for the PhD proposal [Zim01] of the first author to appear.}
share knowledge in a multi-agent system. It is based on the *speech act theory* developed by Austin [Aus62] and Searle [Sea69]. The primitives of KQML are called *performatives* which define the permissible "speech acts" that are allowed to perform in communication with each other. Thus, KQML messages do not solely communicate sentences in some language, but rather communicate an attitude about the content of the message. Typical KQML performatives are, e.g. *ask*-if, or *tell* with which an agent can ask another agent whether a given formula is valid in its VKB, or tell the other agent to add a fact to its KB, respectively.

The exchange of several KQML messages between one or more agents about a certain topic is called a *KQML conversation*. Usually, KQML conversations must obey a strict protocol which specifies valid conversations. The validity of an incoming message is checked by the *conversation module* of an agent. If an agent receives a message that cannot be assigned to an ongoing conversation, it simply ignores this message. LABROU defines a basic set of valid conversations in [Lab96].

KQML *facilitators* are special agents which fulfill tasks as, e.g. *message* forwarding and broadcasting, mapping of symbolic agent names to real physical addresses, and managing a list of the capabilities of all known agents (see [Lab96] for further details).

### 3.2 MathWeb Agents

We followed the ideas of AOP and developed a prototype agent shell for mathematical services. Fig.3 shows the general structure of MathWeb agents as they are currently realized to build the bridge between the MathWeb-SB and the LBA. Essentially, each agent contains a classical reasoning or computation service. A central control unit composes and sends KQML messages and translates incoming messages into concrete actions for the service. Each MathWeb agent can access a database of mathematical services specifications in order to produce reasonable KQML messages. The conversation module of an agent handles all ongoing conversations the agent is involved in. MathWeb agents are supposed to communicate via KQML with OPENMATH as a content language. In the future, we might also use DLOG [Koh00] or other languages as content languages. While OPENMATH is a standard for the representation of formulas, DLOG aims at the representation of full mathematical documents, including, e.g. plain text, definitions, theorems, and proofs. OPENMATH and DLOG are based on a fixed ontology accepted by many members of the automated reasoning and of the computer algebra community. Therefore, we think that OPENMATH and DLOG are more suitable as content languages for the communication between MathWeb agents than other languages (e.g. KIF or MathML).
3.3 The Bridge

The key idea behind the bridge between the MATHWEB-SB and the LBA is to wrap existing mathematical services into a MATHWEB agent shell and to extend both systems by KQML facilitators which can communicate via a common interface. The overall structure of our bridge is depicted in Fig.4. We decided to use the Hyper Text Transfer Protocol (HTTP) as the basic communication protocol between the MATHWEB-SB and the LBA because it is light-weight, ubiquitous in the Internet, and there are already implementations of HTTP available for all standard programming languages. The content of an HTTP request is expected to be the XML encoding of a KQML message\(^3\) with an XML-encoded OPENMATH formula as its content. To send a KQML message over the bridge, an HTTP client connects to an HTTP server and sends a request with the message as content. HTTP clients and servers are provided by so called routers. Routers also perform the translation of the internal representations of KQML messages into the XML format and vice versa. We implemented two HTTP routers, one in the MATHWEB-SB (implemented in MOZART) and one in the LBA (implemented in Java).

![Figure 4: The bridge between MATHWEB and LBA](image)

We now describe the additional extensions of the MATHWEB-SB and the LBA that were necessary to build the bridge.

**Extension of the MathWeb-SB:** Since the MATHWEB brokers already offer features like the registration of services and the forwarding of requests we decided to extend them to full KQML facilitators. Of course, these extended brokers do not lose any of their capabilities.

Furthermore, we implemented the MATHWEB agent structure proposed in section 3.2 in MOZART and wrapped the OM\(\text{EGA}\) and the MBASE system into a MATHWEB agent shells. Fig.4 shows the OM\(\text{EGA}\) agent \(O\) and the MBASE agent \(M\) and the LBA agent \(L\) which play a central role in the case study described in section 4.

\(^3\) The document type definition (DTD) of KQML is available at [http://www.mathweb.org/mathweb/kqml/dtd/kqml.dtd](http://www.mathweb.org/mathweb/kqml/dtd/kqml.dtd).
Towards Mathematical Agents – Combining MATHWEB-SB and the LBA

We expect, that in every local agent society, there is at least one KQML facilitator and that all MATHWEB agents in this society communicate directly with this facilitator. The facilitators do the routing of outgoing messages and forward incoming messages to the receiver. For instance in Fig.4 the ÔMEGA agent communicates only with the local facilitator F1. When sending a KQML message to an agent outside of MATHWEB the facilitator F1 accesses the router R1 that translates the message (including the OPENMATH content) from the MOZART-internal representation into XML encoding. When R1 receives a message via an HTTP request it parses the message and the content and translates it into the MOZART representation.

Extension of the LBA: We also extended the LBA by a KQML facilitator (F2) (which can access the router R2) and by one central LBA agent L which can access all logical services of the LBA. Fig.4 shows up a slight asymmetry between the agent structure in the MATHWEB-SB and the LBA. This is due to the fact that the two logic services (factor and simplify) currently offered by the LBA do not actively send subproblems to other MATHWEB agents outside the LBA. The use of one agent for the whole LBA allows us to keep the LBA in its current state and to perform agent communication. The extensions made on the LBA side are located within the dotted region in Fig.4. Consequently, the KQML facilitator (F2) is not identical with the Logic Broker. The LBA agent L acts as a Logic Broker client. When a reasoning service of RDL is required, agent L subscribes to the service registered by the RDL service server and receives a CORBA service object (cf. section 2.2). The agent can then send the problem to be solved to this service object.

The extension of the MATHWEB-SB and the LBA presented in the previous paragraphs fulfill all our requirements. They preserve the original functionality of both systems and allow to run the systems with or without MATHWEB agents and KQML communication. The interaction protocol between the two architectures is based on widely used standards. Using our bridge, the ÔMEGA system can already access the services factor and unsat provided by the LBA. Every external system which wants to use the reasoning services integrated in the MATHWEB system or in the LBA can easily access them. For this, the external system must use an HTTP client and server which sends appropriate KQML messages and interprets incoming messages. This is sufficient to perform simple queries, but in order to perform full agent communication the external system should also be able to handle full KQML conversations.

4 Case Study

We now describe a case study which shows that the bridge proposed in the previous section is sufficient for communication between MATHWEB agents and that an interplay between reasoning systems can be modeled as a KQML conversation. We intend to use the reasoning capabilities of RDL and MBase within the proof planner of the ÔMEGA system.

RDL is based on CCR (Constraint Contextual Rewriting) and simplifies clauses in a quantifier-free first-order logic with equality using a tight integration between rewriting and decision procedures. In its current version, RDL offers the decision procedures la for the theory of Universal Presburger Arithmetic over Integers (UPAI), eq for the Universal Theory of Equality (UTE), and aug(la) for UPAI extended with uninterpreted function symbols.

4 However, this might change in a future version of the LBA.
MBASE is a knowledge base for mathematical objects, such as definitions, theorems, or proofs. It can serve as a shared background for various deduction services. Its object representation is fully based on OPENMATH and ODOC. Since it establishes a semantics for the objects, these systems reason about, it can serve as the necessary “glue” that binds mathematical services together to a joint system. On the other hand, the formal representation allows semantics-based retrieval of distributed mathematical facts.

In the following section we describe the problem of instantiating meta-variables\(^5\) in planning proofs for limit theorems. In section 4.2, we show how RDL’s decision procedure \(\text{aug(ia)}\) can be used as an oracle to find appropriate instantiations for the meta-variables. Finally, we show in section 4.3 how the interaction between \(\Omega\)MEGA, RDL, and MBASE can be modeled as a KQML conversation between MATHWEB agents.

4.1 The Problem

In recent years we gained some experience in using knowledge-based proof planning [MS99] to find \(\varepsilon\)-\(\delta\)-proofs for limit theorems. Knowledge-based proof planning is a variant of proof planning introduced by BUNDY [Bun88] which makes extensive use of mathematical knowledge. Due to space limitations we don’t go into the details of proof planning here. The interested reader should consider [Bun88], [KK98], and [MS99] for further information.

One of the main problems in proof planning in general is the instantiation of meta-variables during the proof planning process or after a complete proof plan has been found. Our first attempt to tackle this problem was the development the constraint solver \(\text{CoSTIE} [\text{Zim00}]\) which is integrated into the proof planner of \(\Omega\)MEGA as an external reasoning system [MMZ99].

In order to plan proofs for certain theorems (e.g. theorems about trigonometric functions) reasoning system is needed which can handle user-defined function symbols and uses an extensible set of facts (e.g. about properties of user-defined functions). These requirements are not met by the current implementation of \(\text{CoSTIE}\). In the following, we will show that (at least in some cases) the decision procedure \(\text{aug(ia)}\) of RDL can be used as an oracle for the instantiation of meta-variables when the constraints for these variables contain user-defined function symbols.

The problem of instantiating meta-variables also occurs when the Squeeze Theorem is applied to an open subgoal. The Squeeze Theorem states that one can prove that the limit of a function \(g\) at a point \(a\) is \(L\) if one can squeeze \(g\) with two functions \(f\) and \(h\) (we also refer to as the squeezing functions) whose limit in \(a\) is \(L\).

Example 1. In order to prove that \(\lim_{z \to 0} (\sin(z) + 3\cos(z)) = 3\) we can apply the Squeeze Theorem with 

\[
g(x) = (\sin(x) + 3\cos(x))
\]

and eventually have to find functions \(F(x)\)\(^6\) and \(H(x)\) who squeeze the given function:

\[
F(x) \leq \sin(x) + 3\cos(x) \leq H(x)
\]

and whose limit at 0 is 3, i.e., \(\lim_{z \to 0} F(x) = 3 = \lim_{z \to 0} H(x)\).

\(^5\) Meta-variables are place holders for (higher order) witness terms.

\(^6\) Here \(F\) and \(H\) are higher order meta-variables that must be instantiated by a \(\lambda\)-expression.
4.2 Computing squeezing functions

We use the RDL system as an oracle which can help to find appropriate squeezing functions. In the following, we expect that the mathematical knowledge base MBASE solely contains some basic facts about the functions \( \sin \) and \( \cos \):

\[
\forall x. -|x| \leq \sin(x) \leq |x|, \tag{2}
\]

\[
\forall x. 1 - x^2 \leq \cos(x) \leq 1. \tag{3}
\]

RDL's decision procedure for linear arithmetic with augmentation (aug(la)) can help to derive a squeezing function \( F(x) \) for inequality (1) from the facts (2) and (3) (see [ACR01] for information about the decision procedure). The decision procedure is designed to detect unsatisfiability of ground inequalities\(^7\), so we provide the negated theorem. For Example 1, we provide

\[
\neg(F(x) \leq \sin(x) + 3 \cos(x))
\]

(4)

to get a candidate for a lower bound squeezing function. RDL first sends a query to MBASE to get all available facts about the user-defined function symbols (\( \sin \) and \( \cos \)) in the input formula. Then, the decision procedure can derive that for unsatisfiability (i.e. validity of the theorem), the following inequality must hold:

\[
3 - 3x^2 - |x| \geq -F(x)
\]

(5)

From inequality (5) we can easily derive an appropriate instantiation for \( F \leftarrow \lambda x. 3 - 3x^2 - |x| \). A similar computation can be performed to get a candidate for the upper squeezing function \( H \).

4.3 The KQML Communication

We now take a closer look at the communication that takes place between the \( \Omega \) agent \( O \), the MBASE agent \( M \), and the LBA agent \( L \) in order to achieve the desired interaction.

For the sake of readability we use a proprietary syntax for KQML messages in the following. We abstract from message IDs and message handling details and use the expression \( P(S, R, C) \) to indicate that agent \( S \) sends a message with the performative \( P \) and the content \( C \) to the receiver \( R \), where we assume that the content \( C \) is always an OpenMath formula. For the same reason we do not use XML syntax for OpenMath formulas but the usual math notation.

In the following, we assume for the sake of brevity that the \( \Omega \) agent \( O \) already knows that the LBA agent \( L \) offers a service \texttt{unsat}(\( F, X \)) which checks whether a given OpenMath formula \( F \) is unsatisfiable. \( X \) is an annotated OpenMath variable which is bound to the result of the agent's computation. The service \texttt{unsat} either returns \texttt{true}, if the given formula \( F \) is unsatisfiable, or a formula which has to be valid in order to prove unsatisfiability.

In the following we refer to Example 1 of section 4.1 and suppose that the application of the Squeeze Theorem produced (among others) the subgoal

\[
F(x) \leq \sin(x) + 3 \cos(x)
\]

(6)

\(^7\) Since RDL can only handle ground terms, we actually have to replace the meta-variables \( F(x) \) and \( H(x) \) in Example 1 by fresh constants. For the sake of readability, we keep using the meta-variables here.
The instantiation strategy of \textsc{Omega}'s proof planner is triggered by a special control mechanism which suggests the call of the \texttt{unsat} service directly after the application of the \textsc{Squeeze Theorem}. The instantiation strategy then uses the \textsc{mathweb} agent shell for the \textsc{kqml} communication, i.e. and agent \texttt{O} starts a new \textsc{kqml} conversation with agent \texttt{L} and asks \texttt{L} to check the unsatisfiability of the negated inequality (6):

\texttt{ask-one(O, L, unsat(-(F(x) \leq \sin(x) + 3 \cos(x)), X))}.

The conversation module of agent \texttt{L} accepts the \texttt{ask-one} message and starts a new thread to handle the conversation. The conversation module also builds up the context of this conversation. In our case the context consists additional facts for the user-defined functions symbols \texttt{sin} and \texttt{cos}. To retrieve these \texttt{L} asks the \texttt{Mbase} agent \texttt{M} for known upper and lower bounds. \textsc{Mbase} can provide two theorems that define upper and lower bounds:

\[
\texttt{tell(M, L, } \forall x. -|x| \leq \sin(x) \leq |x|,)
\]
\[
\texttt{tell(M, L, } \forall x. 1 - x^2 \leq \cos(x) \leq 1).
\]

Finally, the \texttt{RDL} agent subscribes to the \texttt{unsat} service available at the \texttt{logic broker} (see Fig.4). The decision procedure of \texttt{RDL} is invoked with the facts stored in the conversation context and with the given formula as input. In our example the decision procedure cannot determine the logical status of the input formula, but after termination it returns the inequality (5) of section 4.2 which has to be valid for the input to be unsatisfiable. Thus the \texttt{openmath} variable \texttt{X} in the \texttt{ask-one} performative is replaced by this inequality and the response of the \texttt{RDL} agent is

\[
\texttt{tell(L, O, unsat(-(F(x) \leq \sin(x) + 3 \cos(x)), 3 - 3x^2 - |x| \geq F(x)))}
\]

The \textsc{Omega} agent can now pass the inequality \(3 - 3x^2 - |x| \geq F(x)\) to the instantiation strategy which replaces the meta-variable \texttt{F(x)} by \(\forall x. 3 - 3x^2 - |x|\). A similar conversation with the \texttt{RDL} agent yields the candidate \(\forall x. 3 + |x|\) for the squeezing function \texttt{H}. Finally, the planner tries to find proof plans for the subgoals \(\lim_{x \to 0} 3 - 3x^2 - |x| = 3\), and \(\lim_{x \to 0} 3 + |x| = 3\). These subgoals can easily been proved with the standard proof planning methods for \(\varepsilon\text{-}\delta\)-proofs.

5 Related Work

\textsc{Oants} [BS00] is an agent-oriented command suggestion mechanism for interactive theorem proving within the \textsc{Omega} system. \textsc{Oants}' suggestion agents autonomously gather information from \textsc{Omega}’s central proof data structure and deliver it to the command suggestion agents. The suggestion agents then suggest appropriate proof steps to a human user. \textsc{Oant} agents do not communicate with a standard \texttt{ACL} but use \texttt{lisp}-internal communication.

\textsc{Netsolve} [ALD00] is a client-server system that enables users to solve complex scientific problems remotely. The system allows users to access both hardware and software resources distributed across a network. In [ALD00] \textsc{Arnold et al.} also describe the linking of \textsc{Netsolve} with other systems (e.g. \textsc{Ninf} and \textsc{Condor}). \textsc{Netsolve} is designed for distributed scientific computing rather than for cooperative mathematical reasoning.

\textsc{The discount} theorem prover [DKS97] is based on the \textsc{Teamwork} approach for distribution problem solving. Experiments with \textsc{discount} explored the aspect of a tight cooperation between the theorem provers that makes a group mathematical services more successful than any single component.
6 Conclusion

We presented our approach for the interconnection of two architectures for distributed automated reasoning, the MATHWEB-SB and the LBA, via an agent-based communication bridge. To build the bridge, we extended the reasoning systems integrated in the MATHWEB-SB and the LBA to full MATHWEB agents which communicate with KQML messages. We also augmented both architectures by added KQML two facilitators that exchange KQML messages using the HTTP protocol.

The proposed bridge preserves all the advantages and functionalities of both architectures and is independent of the underlying implementation platforms. The bridge allows two reasoning systems to perform cooperative problem solving even if they are located within the MATHWEB-SB and the LBA respectively. Last but not least, the common protocol introduced in this paper can also be used by external applications to access the services offered by both systems.

7 Acknowledgments

We would like to thank Silvio Ranise for his help in developing the RDL server and Andreas Franke for his support for MBase. We are also indebted to Andreas Meier for his help on the proof planning part.

References


Discovering Theorems using GOEDEL:
A Case Study

Johan Gisbertus Frederik Belinfante

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

Abstract Combining an interactive symbolic manipulation program with
a theorem prover allows one to discover theorems as well to prove them.
The specific focus in this paper is on illustrating how to use the GOEDEL
program, a Mathematica™ implementation of Gödel's algorithm for
class formation, to help discover theorems about sets satisfying some
property hereditarily. Similar techniques are applicable to other topics
in set theory. Formal proofs of many of these theorems have been ob-
tained using McCune's first order automated reasoning program Otter.

1 Introduction

In this paper we focus on illustrating how a symbolic manipulation program can help
discover theorems in set theory. This is part of ongoing work (Belinfante, [Bel99a],
[Bel99b] and [Bel00b]) on proving theorems of NBG class theory, building on prior
work by Robert Boyer et al. ([Boyeretal]) and Art Quaife ([Quaife92a], [Quaife92b]),
using McCune's ([McCune94]) first order logic automated reasoning program Otter.

Our aim is not to announce stunning new theorems, but rather to explain the fairly
prosaic techniques that were found to be useful. Further details, including related the-
orems, proof summaries and other pertinent information about each of the Otter proofs
of the theorems mentioned here, as well as for several thousand other theorems in set
theory, may be found on the author's website: http://www.math.gatech.edu/~belinfan/research/
A recent version of the GOEDEL program is also provided there.

The theorems to be discussed here are about sets satisfying some property heredi-
tarily. For example, a set is said to be hereditarily finite if it is not only finite, but all
its members are finite, and all their members in turn, and so on. Another example of
interest concerns a characterization of the class $\Omega$ of all ordinal numbers. It is well-
known (Monk [Monk69]) that when the axiom of regularity holds, the class of ordinal
numbers can be described as the class of hereditarily full sets. It was discovered that
something similar (Theorem H-ON-3) holds without assuming the axiom of regularity.

2 Brief Description of the GOEDEL program

The GOEDEL program, originally developed (Belinfante, [Bel96] and [Bel00a]) to help
prepare input files for proofs in set theory using McCune's automated reasoning pro-
gram Otter, implements in Mathematica™ an algorithm Kurt Gödel ([Goedel40])
presented in his proof of his basic class existence metatheorem schema.

Although the program was named after Gödel's finite axiomatization for class the-
ory, the main ideas are older and stem from Bernays's reformulation ([Bern37]) of von
Neumann's class theory. By syntactically analyzing the structure of a statement \( p(x) \), the usual class formation constructor \( \{ x \mid p(x) \} \) can be eliminated in favor of building classes in terms of a finite number of primitives, for example, the universal class \( V \), the membership relation \( \in \) and seven other basic class constructors: complement, domain, flip, rotate, pairset, cart, intersection. Bernays ([1981], page 64) points out that even fewer could be used.

Since the \textsc{Gödel} program itself has already been described elsewhere (Belinfante, [Bel96] and [Bel00]), only a brief indication of its general features will be given here. Some familiarity with Mathematica is assumed on the part of the reader. At the heart of the program is the algorithm for converting the customary definitions of classes into expressions built up from the primitive constructors. This algorithm is presented in the \textsc{Gödel} program as a series of definitions for a Mathematica function \texttt{class[x,p]}. The first argument \( x \), which is assumed to be a set, must be either an atomic symbol, or an expression of the form \texttt{pair[u,v]} where \( u \) and \( v \) in turn are either atomic symbols or pairs, and so on. The second argument \( p \) is a statement which may involve the variables that appear in the expression \( x \), as well as other variables that may represent arbitrary classes (not just sets). The statement may contain quantifiers, but all quantified variables must be sets. The quantifiers \texttt{forall} and \texttt{exists} used in the \textsc{Gödel} program are explicitly restricted to set variables.

Many rewrite rules must be added to produce compact definitions. With these added rewrite rules, Gödel's proof of termination no longer applies, but the \textsc{Gödel} program nonetheless has proved to be a practical ad hoc tool for formulating definitions and for simplifying statements of theorems.

Although it contains no systematic mechanism for carrying out deductions, the \textsc{Gödel} program does sometimes manage to prove statements by simplifying them to \texttt{True}. In addition, some rudimentary tools have been developed to coax the \textsc{Gödel} program to reveal new facts, as will be explained shortly.

### 3 A simple example and some history

From a user's standpoint, a most useful feature of the \textsc{Gödel} program is its ability to recognize that various different specifications of a class are equivalent. This use of \texttt{class} is illustrated by the following different descriptions of the class \texttt{FULL} of full sets.

\[
\text{class[x,forall[y,z,implies[and[member[y,z],member[z,x]], member[y,x]]]]} = \text{FULL}
\]
\[
\text{class[x,forall[y,implies[member[y,x],subclass[y,x]]]]} = \text{FULL}
\]
\[
\text{class[x,subclass[U[x],x]]} = \text{FULL}
\]
\[
\text{class[x,subclass[x,P[x]]]} = \text{FULL}
\]
\[
\text{class[x,full[x]]} = \text{FULL}
\]

Here and elsewhere in this paper, for the sake of brevity, an equation \( a = b \) is written to indicate that Mathematica input \( a \) produces output \( b \) for some version of the \textsc{Gödel} program. As facts discovered with one version are added as rewrite rules in later ones, the \textsc{Gödel} program is continually evolving.

Consider for example the following two examples involving \texttt{class}:

\[
\text{class[x,exists[y,and[member[x,y],full[y]]]]} = \text{U[FULL]}
\]

and
class[x, exists[y, and[subclass[x, y], full[y]]]]
    = image[inverse[S], FULL]

The current version of the GOEDEL program simplifies both of the above two expressions
to the universal class V. One would have to go back to a version of the GOEDEL program
as it was on 1999 February 12 to get the results stated above, because on the following
day the rewrite rule

    image[inverse[S], FULL] := U[FULL]

was added. An Otter proof of this equation had just been found on that day; this is
Theorem FULL-MS2 in the FULL\3 group. The proof is not complicated; it just uses the
facts that if a set x is full, then so are the sum set U[x] and the power set P[x]. Although
the equation U[FULL] = V had long ago been proved by hand, it was not added as a
rewrite rule to the GOEDEL program until 2000 October 16, when an Otter proof of
this equation was obtained. This proof found by Otter uses transfinite induction and
a recursive definition of transitive closure.

The rewrite rules in the GOEDEL program can simplify statements as well as
descriptions of classes, and in particular, can be used to eliminate quantifiers. Given any
statement p, one can form the class class[w, p] where w is any variable that does not
occur in the statement p. This class is the universal class V if p is true, and is the empty
class when p is false. The Mathematica definition

    assert[p_] := Module[{w = Unique[]}, equal[V, class[w, p]]]

thus produces a new statement equivalent to the original one. The occurrence of class
here causes Gödel's algorithm to be invoked, the meaning of the statement p to be
interpreted, and the rewrite rules in the GOEDEL program to be applied. The transformed
statement need not be simpler than the statement one started with, but often it is. To
improve readability of the output, further rewrite rules are sometimes used to convert
the equations obtained with assert back to simpler nonequational statements.

4 Membership Rules

When one wants to extend the GOEDEL program to deal with a new concept, the first step
is to add an appropriate membership rule. For example, for the power class constructor
P[x], the GOEDEL program contains the membership rule

    member[x, P[y_]] := and[member[x, V], subclass[x, y]]

Another example is the definition of the class FULL of all full sets:

    member[x, FULL] := and[member[x, V], subclass[U[x], x]]

Whenever a membership rule involves quantifiers, it is the author's practice to
wrap it with class to force those quantifiers to be eliminated. A typical example is the
following wrapped membership rule which is used to define the sum class constructor
U[x]:

    class[w, member[x, U[z_]]] := Module[{y = Unique[]},
        class[w, exists[y, and[member[x, y], member[y, z]]]]]

A second example is the definition of the subset relation S.
In this case one needs variables in order to rewrite \( x \) as an ordered pair. It should be pointed out that in the \texttt{GOEDEL} program it is explicitly assumed that all quantified variables must refer to sets, and not to proper classes. For this reason it is not necessary to add \texttt{member[u,v]} and \texttt{member[v,u]} on the right side of this membership rule. For the benefit of Mathematica experts we remark that the wrapped membership rules must nonetheless be given as downvalues for \texttt{class}, and not as upvalues for \texttt{member}; attempting to do the latter would cause looping to occur.

In the case of the subclass relation \( S \) the \texttt{GOEDEL} program also contains another unwrapped membership rule

\[
\text{member[pair[u,v],S]} := \\
\text{and[member[u,v],subclass[u,v]]}
\]

which applies only to ordered pairs. In this rule, it would have been safe to omit \texttt{member[u,v]}, but it would not be correct to omit \texttt{member[v,u]}. To rule out the possibility that adding such a second membership rule might lead to an incompatible definition for a particular class, it is wise not to add such a rule until it has been formally proven to be correct, either by using \texttt{Otter} or by using the \texttt{assert} mechanism of the \texttt{GOEDEL} program itself to discover the second rule.

5 Normalization Rules

Any class \( x \) is the class of all its members, a fact which is expressed in the \texttt{GOEDEL} program as a default rule:

\[
\text{class[u, member[u,x]]} := x /; \text{And[FreeQ[x,u],AtomQ[u]]}
\]

This default rule must be placed after all specific wrapped membership rules in order for the specific rules to have a chance of being applied. This feature has one annoying side effect of which the user should be aware: one cannot just add wrapped membership rules on the fly in a Mathematica session, because Mathematica's built-in precedence rules for evaluation will cause the default rule to be applied instead of the rule that one has just added.

If the class \( x \) is replaced with some specific class, then it is as likely as not that \texttt{class[u, member[u,x]]} will not simplify to \( x \), a fact that one can work to one's advantage. If this expression does simplify to the same \( x \) with which one started, we say that the class \( x \) is normalized. Whether an expression is normalized or not depends of course on what simplification rules are present. Whatever this expression does simply to is in any case equal to the original expression \( x \), thereby sometimes enabling one to discover an alternative formula for a particular class of interest.

To expedite this type of discovery, a separate file \texttt{TESTS.\!\!} containing various useful Mathematica definitions such as

\[
\text{Normalize[x]} := \text{Module[{w=Unique[]},member[w,v]=True; class[w,member[w,x]]]}
\]

\[
\text{Normality[x]} := (x == \text{Normalize[x]})
\]
may be loaded along with the \texttt{GOEDEL} program itself. This file also contains variants of Normality in which one or two assert's have been wrapped around member[v,x]. These variant tests are called Renormality and Basenormality, respectively. The beauty of these tests is that one can specify exactly for which class a simpler formula is to be sought.

For binary and ternary relations, variants have been added which can exploit the membership rules that involve ordered pairs or ordered triples. The simplest of these is:

\begin{verbatim}
ReInNormality[x_] := Module[{u=Unique[], v=Unique[]},
    member[u,v]=True; member[v,v]=True;
    Equal[composite[Id,x],
    class[pair[u,v],member[pair[u,v],x]]]]
\end{verbatim}

Again there are also variants with assert wrappers. Experience indicates that what works even better for many binary and ternary relations are still further variants which rely on vertical section rules instead of membership rules. The bare bones rule is called VSNormality:

\begin{verbatim}
VSNormality[x_] := Module[{u=Unique[], v=Unique[]},
    member[u,v]=True; member[v,v]=True;
    Equal[composite[Id,x], class[pair[u,v],
    member[v,image[x,singleton[u]]]]]
\end{verbatim}

Typically one barrages a relation of interest with a whole battery of such tests to flush out any interesting formulas that might prove useful.

When one does succeed in discovering some useful formula, one may add the new rule as a permanent new simplification rule, thereby enhancing the power of the \texttt{GOEDEL} program. Adding such rules has the beneficial effect of forcing simple expressions to simplify to themselves. One can of course take the first expression that comes along to enforce normalization, but doing so could lead to rather complicated rules of little intrinsic interest. In a few cases where one may have to resort to the expedient of adding rather complicated normalization rules, one may still hope that later on some simpler rules would be discovered. Our experience indicates that adding a complex normalization rule is better than adding no rule at all, and one often learns later how to break up complex rules into smaller pieces even after having temporarily put some complicated rule in place.

Since the \texttt{GOEDEL} program is to be used for discovery rather than proof, we do not hesitate to add facts proved using \texttt{Otter}. To avoid subtle errors, however, we avoid adding rules for which only a proof by hand is available because humans tend to gloss over uninteresting details, such as the requirement that some set be nonempty, or that some relation be thin. To avoid circularity, theorems discovered using with the \texttt{GOEDEL} program are never added to the usable list in \texttt{Otter}. Such facts are regarded as conjectures; of course, even mechanized proofs do not preclude the possibility of error (Belinfante [Bel97]).

Finding formal proofs with \texttt{Otter} is generally more challenging than using \texttt{GOEDEL} to discover new facts to be proved. For this reason, the \texttt{Otter} proofs tend to lag behind what has been discovered using the \texttt{GOEDEL} program. Nonetheless, the \texttt{GOEDEL} program is no substitute for \texttt{Otter} because it does not produce readable proofs. One can of course try to find out how \texttt{GOEDEL} did its work by using Mathematica's built-in \texttt{Trace} procedure. This sometimes works, but often as not this yields a voluminous and rather unintelligible accounting of what took place.
6 A lambda calculus

A principal obstacle in using a first order theorem prover that relies on the clause language is that the Skolem functions that are introduced in the process of converting statements to clause form often lead to expressions of high weight. To avoid frequent intervention by hand to cope with this problem, it is often desirable to eliminate quantifiers over set variables. Minimizing the number of Skolem functions that need to be introduced often allows one to reduce the number of clauses needed, as well as the number of literals in a given clause, thereby greatly improving the readability of the statements of theorems and of their proofs. An important tool for accomplishing this is to introduce whenever possible bonaﬁde set-theoretic functions corresponding to the function symbols of ﬁrst order logic. Thus, for example, in addition to introducing the function symbol \( tc[x] \) for the transitive closure of a class \( x \), it is useful to introduce also the related set-theoretic function \( TC \) whose members are the ordered pairs \( \langle x, tc[x] \rangle \), where \( x \) is any set.

The basic constructor \( \text{VERTSECT} \) provides a standard way to obtain deﬁnitions for many functions. This enables one to deﬁne functions by specifying the result obtained when they are applied to an input. The basic idea is not limited to functions; any relation can be speciﬁed by giving a formula for its vertical sections. The vertical sections of a relation \( z \) are the family of classes

\[
\text{class}[y, \text{member}[(x, y), z]] = \text{image}[z, \text{singleton}[x]].
\]

Once one comes to realize the usefulness of working with vertical sections instead of points, it becomes natural to introduce the function which assigns these vertical sections:

\[
\text{class}[(x, y), \text{equal}[(y, \text{image}[z, \text{singleton}[(x)])]] = \text{VERTSECT}[z]
\]

Gödel’s algorithm ﬁrst converts the left side of this formula to the expression

\[
\text{composite}[(x, y), \text{intersection}[
\text{complement}[(x, \text{composite}[(x, \text{complement}[(x)])]),
\text{complement}[(x, \text{complement}[(x, \text{complement}[(x)])])]]].
\]

What happens after this depends on \( z \). If the class \( z \) is left unspeciﬁed, various normalization rules will simply convert this expression to \( \text{VERTSECT}[z] \). If the class \( z \) is known to be a function with domain \( V \), a different set of simpliﬁcation rules will be applied, and one will obtain the function \( \text{composite}[(\text{SINGLETON}, z)] \). If \( z \) is the inverse of the membership relation \( E \), one obtains \( \text{Id} \), and so on.

For many relations \( z \) the vertical sections need not be sets. The domain of \( \text{VERTSECT}[z] \) in general is the class of all sets \( z \) for which the vertical section \( \text{image}[z, \text{singleton}[x]] \) is also a set. Let us call a relation \( th \) when all its vertical sections are sets. The axiom of replacement implies that functions are thin, and the sum class and power class axioms imply that \( \text{inverse}[E] \) and \( \text{inverse}[S] \) are thin, where \( E \) and \( S \) are the membership and the subset relations, respectively.

One can use \( \text{VERTSECT} \) to ﬁnd a formula (Belinfante, [Bel00a]) for any function from a formula for its application \( \text{image}[x, \text{singleton}[x]] \). This is done neatly in the \( \text{GODEL} \) program by introducing the Mathematica deﬁnition

\[
\text{lambda}[x, e] := \text{Module}[(y=\text{Unique[]}], \text{VERTSECT}[\text{class}[(x, y), \text{member}[(y, e)])]]
\]
In addition to \textsc{Vertsect}, it is convenient to introduce a related constructor \textsc{Image}, defined by

\begin{verbatim}
lambda [u, image[x,u]] =
\textsc{Vertsect} [composite[x, inverse[u]]] = \textsc{Image}[x].
\end{verbatim}

The constructor \textsc{Image} does not in general preserve composites, but this does hold when the right hand factor is thin. While \textsc{Image} preserves the global identity function, in general \textsc{Image}[id[x]] is not an identity function, but it is nonetheless a useful function. From the \textsc{Goedel} program one learns:

\begin{verbatim}
lambda [w, intersection[x, w]] = \textsc{Image}[id[x]].
\end{verbatim}

\section{Constructions for $H[x]$ and $tc[x]$}

To illustrate how the \textsc{Goedel} program is used, two constructors that recently have been the subject of investigation will be discussed. The classes $H[x]$ and $tc[x]$ are respectively the largest full subclass of $x$ and the smallest full class which contains $x$. Both of these descriptions do eventually emerge as theorems, but one must start with somewhat less transparent definitions. The problem here is that until some construction is given, it is not a-priori clear that a largest full subclass of $x$ exists, nor that there is any smallest full class which contains $x$. In the \textsc{Otter} work, one begins in each case with a formula for a recursive construction of the class.

Intuitively, the transitive closure $tc[x]$ of a class $x$ is the union of $x$, $U[x]$, $U[U[x]]$, and so on. When $x$ is a set, one might entertain the idea of using recursion to define these iterated sum classes, and then defining $tc[x]$ as the sum class of the class whose members are $x$, $U[x]$, $U[U[x]]$, etc. This naive approach will not work when $x$ is a proper class because in that case all of its iterated sum classes are also proper classes, and thus cannot be members of any class. For this reason, it is better to use recursion to construct a relation whose vertical slices are these iterated sum classes, and then to define $tc[x]$ as the range of this relation. (For technical reasons, a slight variant of this was used in which the vertical slices are the partial sums $x$, $\text{union}[x, U[x]]$, etc., but the basic idea is still the same.) The solution of the recursion equation for this relation is obtained as the union of partial solutions by analogy with standard proofs of the recursion theorem.

In this paper the focus is on what has been done since the author's work on transitive closure, so we concentrate mainly on $H[x]$. Consequently little will be said here about how various rules about $tc[x]$ were discovered using the \textsc{Goedel} program, and just discuss how those rules were then used to find out facts about $H[x]$.

One can construct $H[x]$ as the union of all full subsets of the class $x$. This definition, which is used in the \textsc{Otter} work makes no explicit mention of transitive closure:

\begin{verbatim}
U[intersection[FULL,P[x]]] = H[x]
\end{verbatim}

Nevertheless, one still needs to use $tc[x]$ to prove that this construction works, and so recursion still enters indirectly. For this reason, the $H$ group of theorems about $H[x]$ must be placed after the $TC$ group of theorems about transitive closure.

The theorems about $H[x]$ listed below were all proved using McCune's automated reasoning program \textsc{Otter}. The theorems are formulated in the clause language, and in particular, \textsc{Otter}'s notation - for negation and $|$ for disjunction is used here. Theorems flagged with an asterisk are (usually) added to the demodulator list. Complete details
of these proofs are posted on the author’s website; the proof summaries can be found there in the H group.

Theorems H-1 through H-6 were the first ones to be proved, first by hand, and then using Otter. Theorems H-1, H-2 and H-6, taken together, assert that H(x) is the largest full subclass of x. For the most part, the proofs Otter found resembled the hand-produced proofs, except for the order; the author had proved Theorem H-6 first, deducing H-4 and H-5 as corollaries, whereas Otter first proved H-4 and used it to prove H-6. The listing below includes some corollaries that were added later.

Under the additional hypothesis that x is a set, Theorem H-4 follows immediately from Theorem FUL-SC-9, one of the theorems about full sets that had been proved in the course of work [Belinfante [Be99b]] on ordinal number theory. Removing this additional hypothesis requires using facts about transitive closure. Otter found a proof of length 15 for Theorem H-4 on level 8.

% U:\H.USE 2001/03/25
list(usable).
% Definition
equal(U(intersection(FULL,P(x))),H(x)). %*DEF-H

% Some examples
equal(intersection(FULL,P(FULL)),H(FULL)). %*H-FULL-1
equal(U(H(FULL)),H(FULL)). %*H-FULL-2
equal(H(OMEGA),OMEGA). %*H-ON-1

% Basic theorems about H(x)
full(H(x)). % H-1
equal(tc(H(x)),H(x)). %*H-1-TC
subclass(H(x),x). % H-2

% Characterization of the class OMEGA of all ordinals
equal(intersection(REGULAR,H(FULL)),OMEGA). %*H-ON-3

% Connections between tc(x) and H(x)
subclass(x,H(y)) | subclass(tc(x),y). % H-SU-TC1
equal(image(inverse(TC),P(x)),P(H(x))). %*H-TC-2
subclass(x,H(y)) | subclass(tc(x),y). % H-MEM-1

% More basic theorems
subclass(x,y) | subclass(H(x),H(y)). % H-3
full(x) | equal(H(x),x). % H-4

% Corollaries of Theorem H-4
equal(H(REGULAR),REGULAR). %*H-4-REG
equal(H(tc(x)),tc(x)). %*H-4-TC

% Idempotent property
equal(H(H(x)),H(x)). %*H-5

% The largest full subclass of x is H(x)
full(x) | subclass(x,y) | subclass(x,H(y)). % H-6
Theorem H-TC-2 above was one of the discoveries made with the GODEL program coming on the heels of related discoveries involving the thin relation inverse[TC]. For example, a few days earlier, a number of membership rules for inverse images had been discovered using assert, among which was the following equation, which was promptly added as a new rewrite rule

\[
\text{assert}(\text{member}(x, \text{image}(\text{inverse}(\text{TC}), y))) = \text{member}(\text{tc}(x), y).
\]

In the session which led to the discovery of H-TC-2, at an early point the VSNormality test had been applied to the function

\[
\text{composite}([\text{BIGCUP}, \text{IMAGE(id[FULL])}, \text{POWER}] = \text{HC}
\]

which takes any set \(x\) to \(H(x)\), yielding a messy expression, whose exact nature need not concern us here.

\[
\text{In[]} : \text{composite}([\text{BIGCUP}, \text{IMAGE(id[FULL])}, \text{POWER}] / \text{VSNormality}
\]

\[
\text{Out[]} = \text{composite}([\text{BIGCUP}, \text{IMAGE(id[FULL])}, \text{POWER}] = \text{mess}
\]

The output equation was turned around and made into a temporary rewrite rule in order to cause the messy expression to be eliminated, should it ever come up again.

Later in that same session, the VSNormality test was applied again, this time to the relation composite([\text{inverse}(S), \text{HC}]. The actual manner in which the formula H-TC-2 popped up can be summarized as follows:

\[
\text{In[]} : \text{Map}([\text{image}(\#, P(P(x)))] & \text{composite}([\text{inverse}(S), \text{BIGCUP}, \text{IMAGE(id[FULL])}, \text{POWER}]
\]

\[
\text{Out[]} = P[H(x)] = \text{image}(\text{inverse}[\text{TC}], P[x])
\]

Before the discovery of Theorem H-TC-2, two other connections between \(H(x)\) and \text{image}(\text{inverse}(\text{TC}), P(x))\) had been proved using Otter. Theorem H-TC-1 is an alternative description of \(H(x)\) using the function TC, and Theorem H-TC-3 solves exercise 9.5 on page 74 in a book by Thomas Jech ([Jech78]).

\[
\text{equal}(U(\text{image}(\text{inverse}(\text{TC})), P(x))), H(x)). \quad \% \text{H-TC-1}
\]

\[
\text{equal}(\text{intersection}(x, \text{image}(\text{inverse}(\text{TC})), P(x))), H(x)). \quad \% \text{H-TC-3}
\]

When Theorem H-TC-2 was added, it became a new demodulator, causing each of these other theorems first to be transformed, and then to be subsumed by equal(U(P(x)), x) and by Theorem H-PC-I, respectively.

\[
\% \text{a demodulator}
\]

\[
\text{equal}(\text{intersection}(x, P(H(x))), H(x)). \quad \% \text{H-PC-I}
\]

If one assumes the axiom of regularity, the converse of Theorem H-PC-I holds:

\[
\neg \text{AxReg} \mid \neg \text{equal}(\text{intersection}(x, P(y)), y) \mid \text{equal}(H(x), y). \quad \% \text{RE-H}
\]

Consequently, assuming the axiom of regularity, one can characterize \(y = H(x)\) as the only solution of the equation
\( y = \text{intersection}(x,P(y)) \).

See for example Jech ([Jech78]).

In the course of proving these basic theorems about \( H(x) \), several additional results were discovered by hand, such as Theorems H-I and H-PC listed below. Otter found a different proof for Theorem H-PC. Unlike the author’s own proof, this proof used facts about transitive closure.

% converse of H-SU-TCI
\(-\text{subclass}(tc(x),y) | \text{subclass}(x,H(y)). \quad \% \text{H-SU-TCI} \)

% the constructor \( H \) preserves intersection and power class
\( \text{equal}(\text{intersection}(H(x),H(y)),H(\text{intersection}(x,y))). \quad \% \text{H-I} \)
\( \text{equal}(H(P(x)),P(H(x))). \quad \% \text{H-PC} \)

% a generalization of epsilon induction
\(-\text{subclass}(\text{intersection}(x,P(y)),y) | \text{subclass}(\text{intersection}(\text{REGULAR},H(x)),y). \quad \% \text{H-REG-SU} \)

% converse of H-MEM-1
\(-\text{member}(x,y) | -\text{subclass}(tc(x),y) | \text{member}(x,H(y)). \quad \% \text{H-MEM-2} \)
end_of_list.

8 Membership rules for \( H[x] \) and \( tc[x] \)

For the GOEDEL program, the constructions of \( H[x] \) and \( tc[x] \) are not used directly, but one does need to add appropriate membership rules.

Theorems H-MEM-1 and H-MEM-2 characterize \( H(x) \) as the class of all elements \( y \) of \( x \) for which \( tc(y) \) is contained in \( x \). This characterization of \( H(x) \) is the basis for the membership rule used to define \( H[x] \) in the GOEDEL program.

To deal with the new functor \( H[x] \), the following membership rule was added:

\( (* \text{ added 2000 March 15 *)} \)
\( \text{member}[x,H[y]] := \text{and}[\text{member}[x,y],\text{subclass}[tc[x],y]] \)

This defines \( H[x] \) in terms of transitive closure \( tc[x] \).

The membership rule used for \( tc[x] \) has a rather long history. In the author’s Otter work he began with a recursive definition for transitive closure in order to prove the basic theorem that the transitive closure of a set is a set, which is obtained as a corollary of the equation \( U[\text{FULL}] = V \). After proving those theorems using Otter, he decided as a shortcut to simply add the statement \( U[\text{FULL}] = V \) to the GOEDEL program. But one still wants the membership rule for \( tc[x] \) to remain valid whether \( x \) is a set or a proper class. For sets, one can characterize the transitive closure as the intersection of all full subsets that contain \( x \). For a proper class, the transitive closure is the union of all transitive closures of its subsets. The existential quantifier involved in such a statement can be eliminated by applying assert, and one then arrives at the following quantifier-free membership rule for \( tc[x] \) currently used in the GOEDEL program:

\( (* \text{ added 2001 March 15 *)} \)
\( \text{member}[x,tc[x]] := \text{and}[\text{member}[x,y],\text{not}[\text{subclass}[P[x],\text{image}[	ext{inverse}[S],
\text{intersection}[\text{FULL},P[\text{complement}[	ext{singleton}[x]]]]]]] \)
9 An application: Theorems about hereditarily finite sets

The HF group contains theorems about hereditarily finite sets. The main goal was to show that H(FINITE) is a model for set theory minus the axiom of infinity. For this one mainly needs to establish that H(FINITE) is preserved by various basic set theoretic constructions that occur in the Gödel axioms. (See Theorem 31 on page 97 in Jech [Jech78])

Much of this work has been done. It is perhaps worth commenting that several of the proofs depend explicitly on Quaife's modification of Kuratowski's construction of an ordered pair. A typical example is the proof of the theorem HF-OP:

subclass(cart(H(FINITE)), H(FINITE), H(FINITE)).

This says that ordered pairs of hereditarily finite sets are hereditarily finite.

% HF.USE 2001/03/27
list(usable).
% all full finite sets are hereditarily finite
subclass(intersection(FULL, FINITE), H(FINITE)). % HF-FUL-1

% natural numbers are hereditarily finite
subclass(omega, H(FINITE)). % HF-QM

% closure under pair, set and union
-member(pair(x,y), cart(H(FINITE), H(FINITE))) |
-member(pairset(x,y), H(FINITE)). % HF-UP

% subsets of hereditarily finite sets are hereditarily finite
-member(x, H(FINITE)) | -subclass(y,x) |
-member(y, H(FINITE)). % HF-SU

% restatement of HF-SU as an equation without variables
equal(image(inverse(y)), H(FINITE), H(FINITE)). %*HF-HER

% closure under power set, cartesian product and singleton
-member(x, H(FINITE)) | member(p(x), H(FINITE)). % HF-PC

-member(x, H(FINITE)) | member(cart(x,y), H(FINITE)) |
-member(cart(x,y), H(FINITE)). % HF-CP

-member(x, H(FINITE)) | member(singleton(x), H(FINITE)). % HF-SS-1

% closure under ordered pairs
subclass(cart(H(FINITE)), H(FINITE), H(FINITE)). % HF-OP

% restatement of HF-PC without variables
subclass(image(Power, H(FINITE)), H(FINITE)). % HF-POW-1

% a demodulator
equal(U(H(FINITE)), H(FINITE)). %*HF-FUL-2

% closure under sum class, domain, range and inverse
-member(x, H(FINITE)) | member(U(x), H(FINITE)). % HF-SC

-member(x, H(FINITE)) | member(B(x), H(FINITE)). % HF-DO

-member(x, H(FINITE)) | member(R(x), H(FINITE)). % HF-RA
\texttt{\textit{member}(x,H(\textit{FINITE})) | member(inverse(x),H(\textit{FINITE}))}. \quad \% \textit{HF-IN}

\% converse of \textit{HF-SS-1}
\texttt{\textit{member}(x,V) | \neg \textit{member}(\textit{singleton}(x),H(\textit{FINITE})) |}
\texttt{\textit{member}(x,H(\textit{FINITE})).} \quad \% \textit{HF-SS-2}

One should always be on the lookout for reformulations of theorems as equations which can be made into demodulators because this helps combat combinatorial explosion. Eliminating variables also helps in this regard because variables can be instantiated in many ways. Two such rules were found here:

\% restatements of \textit{HF-SC} and \textit{HF-PC} as demodulators
\texttt{equal(image(\textit{BIGCUP},H(\textit{FINITE})),H(\textit{FINITE})).} \quad \%\textit{HF-BC}
\texttt{equal(image(inverse(\textit{POWER}),H(\textit{FINITE})),H(\textit{FINITE})).} \quad \%\textit{HF-POW-2}
\texttt{end_of_list.}

Summaries of the proofs of all these theorems about hereditarily finite sets proved using \textit{Otter} are posted in the \textit{HF} group on the author's website.

\section{Outlook and Conclusions}

Mathematicians like to downplay the role of axiomatic set theory, but the plain fact is that there are few areas in modern mathematics where sets can be avoided, and to do so would artificially restrict mathematical practice. For automated reasoning programs to find favor in mathematical research, it is desirable that they can cope with set theory on a routine basis. Fortunately, much progress has been made toward mechanizing set theory, especially by Larry Paulson and his coworkers ([Noel], [Paulson:Gr]), the Mizar group ([Rudn:Tryb]), Formisano and Omodeo ([Form:Ono]), Megill ([Megill]), Farmer ([Farmer]) and others, using diverse methods and axioms. The specific techniques discussed in the present paper are to be sure limited to the NBG formulation of set theory. The elimination of variables using \textit{assert}, for example, would not have been possible without using proper classes. Because the NBG formalism is a conservative extension of \textit{ZF} set theory, one may nonetheless view this formalism in principle (Kunen [Kunen80]) as just a particularly convenient notation.

Automated reasoning involves more than merely finding computer proofs of known theorems. In addition to proving theorems, other activities involved in mathematical reasoning also need to be automated, including the formulation of concise definitions, the discovery of useful theorems, the decision as to which equations should be made into rewrite rules and the organization of large collections of theorems. The \textit{GOEBEL} program is an example of a tool that is proving to be an indispensable companion to \textit{Otter} for proofs in set theory.

We have mentioned just a few of the more useful techniques currently being exploited to discover new theorems using the \textit{GOEBEL} program. Space does not permit discussing other useful techniques, for example, tools based on controlling the order in which rewrite rules are to be applied. By comparing different orders of evaluation, new equations can be discovered, in the spirit of the familiar Knuth-Bendix completion procedure. As yet no serious attempt has been made to unify \textit{GOEBEL} with \textit{Otter} or another reasoning program to produce a unified proof environment, but this would certainly be worth doing.

While promising, our present abilities to prove mathematically interesting theorems in set theory using automated proof techniques still fall far short of what a skillful
mathematician can do by hand. But if the enterprise succeeds, our collective efforts will have accomplished something extraordinarily useful, namely the construction of tools for mechanizing deduction in an area central to all of mathematics. The repercussions would be nothing short of revolutionary.

References

A ‘Calculemus–approach’ to high-school math?

Walther A. Neuper

HTBLA, Höhere Technische Bundeslehranstalt
A-5020 Salzburg, Itzlinger Hauptstr. 30

Abstract A ‘Calculemus-approach’ in constructing software for high-school math is considered, i.e. the question how to encompass logical rigor as well as calculational power by combining concepts and components of theorem provers and of algebra systems for educational use. The requirements for student-users are formulated following recent work done at IST, TU Graz, in close contact to RISC Hagenberg, and based on a prototype originating from this work. The paper presents work in progress towards these requirements w.r.t. four concepts: the logical base, stepwise problem solving, reactive user-guidance, and automated generation of explanations.

1 Introduction

‘Both Deduction Systems and Computer Algebra Systems are receiving growing attention from industry and academia. . . . Mathematical Software Systems have been commercially very successful. Their use is now wide-spread in industry, education, and scientific contexts: there are now literally millions of installations of computer algebra programs.’¹ The author teaches math in ‘notebook-classes’ at an Austrian high-school, where each student has his or her own notebook, and uses it (i.e. an algebra system, mainly) in math, too. The experience with the algebra systems is doubtful: they are not designed for education ²: ‘. . . there is still need for improvement as many application domains still fall outside the scope of existing Deduction Systems and Computer Algebra Systems.’²

In this paper we will discuss a mathematics-engine for tutoring, which necessarily falls outside the scope of existing systems, if the following requirements should be met:

1. solve problems of ‘applied math’ automatically from a given (prepared by an author, and hidden) formalization and specification; if nothing is given, support interactive formalization and specification
2. present calculations close to paper and pencil work, and additionally show their structure, and relations to underlying knowledge
3. allow one to trace a calculation down to elementary steps, i.e. the application of theorems to a proof-state

¹ http://www.cs.unitn.it/~rseba/calculatus2001/meeting.html#generalities
² There is one important and successful exception, mathXpert [Bee4]; and because there is no space for related work, the main difference is the following: mathXpert does not provide explicit specification and thus doesn’t handle subproblems, . . . etc. as proposed in sect.3.2.
(4) allow the user to do all the steps himself or herself, give feedback, and return to a
demonstration-mode on request
(5) generate explanations automatically on request by the student, by a kind of 're-
fection' (and not by an author trying to foresee all possible requests)

The feasibility of the requirements w.r.t. interactivity has been shown by a prototype 
, developed at the Institute of Software Technology, TU Graz 4 in close contact with the
Research Institute of Symbolic Computation, Linz 5 . The interactivity relies on a math-
engine; the latter is the concern of this paper. Two concepts of the math-engine seem to
be novel ones, the hierarchy of problem-types (see 3.2) and the 'reverse rewriting' (see
3.3). In order to motivate the overall design, we want to present all essential concepts,
and we want to mention parts still open within this work in progress; thus, in order to
respect space limits, some of the presentations are sketchy.

2 Logical foundations

If we want to give feedback to a student whether a step in a calculation is admissible
or not, we have to maintain some kind of proof-state, and thus are concerned with the
logical framework. As a consequence, the implementation of the prototype is based on
a theorem prover, and not on an algebra system. The choice between Reduce/Redlog
[DS96] and Isabelle [Pau94] was made in favor of the latter. The main reason was,
Isabelle has already implemented most of the knowledge for high-school math in high-
order logic (HOL), and the respective theories are being developed rapidly.

2.1 Foundations in constructive mathematics

Isabelle aims at proving theorems, whereas traditional high-school math is 'applied
math', mainly concerned with 'example construction problems' [Buc94], where some
objects $x$ are given, meeting some precondition $\eta(x)$, and some objects $y$ are to be
constructed, meeting the postcondition $\rho(x, y)$.

Combining Isabelle's non-constructive concept with the constructive one necessary
for high-school math is likely to be neither harmless nor trivial: The foundations for
the educational system mathXpert [Bee85] have been researched carefully - [Bee88]
demonstrated, that carelessly combining the axiom of extensionality, the recursion
theorem and the separation axiom leads to an inconsistent system. The author is not
aware of research, indicating what problems may arise from the combination of HOL
with set construction as described subsequently.

2.2 Implementation and manipulation of a proof-state

A proof-tree represents a (partially) completed proof. In order to have a simple mapping
between the proof-tree and the external representation of a calculation, the safe LCF-
grounds have been left, implementing the following (where $[]$ denote lists):

**Definition 1.** The set $P$ of proof-trees is inductively defined on nodes
$N = (O, Bs)$ where $O$ is called a *proof-object* and $Bs$ is a list called the
branches of $O$:

3 http://www.ist.tu-graz.ac.at/research/edu/isac/
4 http://www.ist.tu-graz.ac.at/
5 http://www.risc.uni-linz.ac.at/
\( (O, []) \in \mathcal{P} \)
\( (O, Bs) \in \mathcal{P} \) where \( Bs = [N_1, \ldots, N_n] \) with \( N_i \in \mathcal{P} \)

There are two types of proof-objects:

- **problem-objects** are records containing all data concerning the specification of an example
- **solve-objects** are records representing the deductive steps, i.e. steps of logical deduction and application of algebraic laws.

These two kinds of objects are called **proof-objects**; their respective fields will be introduced as soon as needed. The tree root is a problem-object, called the root-problem.

The types of branches model the structures of high-school math, the most specific ones concern set construction, e.g. the calculation

\[
\{1, 2, 3, 4, 5, 6, 7\} \cap \{m \in \mathbb{N}. m//7\} = \{1, 7\}
\]

where \( \divides \) denotes 'divides' in the natural numbers, is represented by a so-called Collect-branch in a solve-object as follows

\[
O.\text{expr} = \{1, 2, 3, 4, 5, 6, 7\} \cap \{m \in \mathbb{N}. m//7\}
\]
\[
O.\text{branch} = \text{Collect}
\]
\[
[O_1.\text{expr} = 1 \in \{m \in \mathbb{N}. m//7\}
O_1.\text{result} = \text{true}
\]
\[
\ldots
\]
\[
O_2.\text{expr} = 2 \in \{m \in \mathbb{N}. m//7\}
O_2.\text{result} = \text{false}
\]
\[
\ldots
\]
\[
O.\text{result} = \{1, 7\}
\]

where \( O.x \) selects the field \( x \) (i.e. \text{expr, branch, result}) from the proof-object \( O \), and where the fields of one and the same \( O \) have the same indent level. The formal definition of this **branch-type** and others is straightforward, and gives the basis of the systems semantics.

The purpose of splitting a simple calculation like that is to get steps, each of which is justified by an elementary theorem proven in Isabelle/HOL. In the example given these theorems could be

\[
\text{def_divisors} \quad \text{divisors } n = \{m \in \mathbb{N}. m//n\} \quad (1)
\]
\[
\text{divisor_leqseq} \quad m//n = m \leq n \wedge m//n \quad (2)
\]
\[
\text{inter_def} \quad \{x. P x\} \cap \{x. Q x\} = \{x. P x \wedge Q x\} \quad (3)
\]
\[
\text{mem_Collect_eq} \quad (a \in \{x. P(x)\}) = P(a) \quad (9)
\]

Then the calculation could be represented on a so-called **work-sheet**, the structure being connected with, and justified by the theorems,

\[
\text{divisors } 7 =
\]
\[
= \{m \in \mathbb{N}. m//7\} =
\]
\[
= \{m \in \mathbb{N}. m \leq 7 \wedge m//7\} =
\]
\[
= \{m \in \mathbb{N}. m \leq 7\} \cap \{m \in \mathbb{N}. m//7\} =
\]
\[
= \{1, 2, 3, 4, 5, 6, 7\} \cap \{m \in \mathbb{N}. m//7\} =
\]
\[
1 \in \{m \in \mathbb{N}. m//7\} =
\]

Rewrite def_divisors (1)
Rewrite divisor_leqseq (2)
Rewrite inter_def (3)
Calculate \( \{m \in \mathbb{N}. m \leq 7\} \) (4)
Check \( \{m \in \mathbb{N}. m//7\} \) (5)
Rewrite mem_Collect_eq (6)
A ‘Calculemus–approach’ to high-school math?

\[
\begin{align*}
= 1/7 \\
= \text{true} \\
2 \in \{m \in \mathbb{N}. m/7\} = \\
= \ldots \\
= \{1, 7\}
\end{align*}
\]

Check 2 (7)
Rewrite mem Collect eq (8)
Collect (9)

where the theorems are applied by the tactic Rewrite, shown flushed right. The labels on the right margin relate to those in the knowledge-base, while the tactics (5), (7), (9) correspond to the meta-logic, represented by the branch-type Collect. These tactics are designed for input by the user and do not indicate the underlying theorems (which is a questionable design decision). The above example exceeds Isabelle’s meta-logic.

A survey on high-school math showed [Nen91a] that virtually all examples can be modeled by three special branch-types like Collect.

The proof-tree \( P \in \mathcal{P} \) is extended by applying a tactic (e.g. Rewrite, Calculate, Check, Collect) to a given formula \( f \) in \( P \), representing a proof-state \( (P, f) \). The tactics semantics is based on the transition relation from \( (P, f) \) to \( (P', f') \), defined for each tactic respectively. These definitions are straightforward, leading to the notions of applicable tactics.

3 Autonomous and interactive problem solving

Having indicated the essential prerequisites for the requirements (2) and (3) on p.92, we turn to the requirement (1) ‘solve problems automatically, or support interactive formalization and specification’.

3.1 The phases of problem solving

are model, specify, solve, and have particular input-data and output-data:

\[
\begin{align*}
description & \rightarrow \ \text{model} \rightarrow \ \text{formalization} \rightarrow \ldots \\
\ldots & \rightarrow \ \text{specify} \rightarrow \ \text{specification} \rightarrow \ldots \\
\ldots & \rightarrow \ \text{solve} \rightarrow \ \text{solution}
\end{align*}
\]

To illustrate we shall use a typical example in high-school math. This example, referred to as ‘maximum-example’ in the sequel, is given by the following description:

Given a circle with radius \( r = 7 \), inscribe a rectangle with length \( u \) and width \( v \). Determine \( u \) and \( v \) such that the rectangle’s area \( A \) is a maximum.

The model phases output is a formalization, potentially in one of the variants

\[
\begin{align*}
F_I & \equiv (\{(r, T), \{A, \{u, v\}\}, \{0 \leq \frac{u}{2} \leq r, \{A = uv, (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2\}\}) \\
F_{II} & \equiv (\{(r, T), \{A, \{u, v\}\}, \{0 \leq \frac{v}{2} \leq r, \{A = uv, (\frac{u}{2})^2 + (\frac{v}{2})^2 = r^2\}\}) \\
F_{III} & \equiv (\{(r, T), \{A, \{u, v\}\}, \{0 \leq \alpha \leq \frac{\pi}{2}, \{A = uv, \frac{u}{2} = r \sin \alpha, \frac{v}{2} = r \cos \alpha\}\})
\end{align*}
\]

This may motivate the definition

\[\text{Rewrite} \text{ applies a rewrite rule once, whereas } \text{Rewrite Set} \text{ applies a (terminating) set of rules}\]
Definition 2. Given a set \( I \) of substitutions, called input-items, a set \( O \neq \emptyset \) of output-variables, and a set \( R \) of relations, the triple \( F = (I, O, R) \) is a formalization if \( (\text{Vars } I) \cap O = \emptyset \).

where \( \text{Vars} \) extracts the first element of each pair in a substitution \(^7\) (and in later use also: \( \text{Vars} \) extracts the variables from a term). The sets within the triple contain different types of objects, in particular sets for the purpose of grouping: a formalization will 'instantiate' a problem-type, see Def.5.

The formalization must be given (by an author, and eventually hidden from the student) in order for the problem to be 'solved automatically'. \(^8\)

3.2 Interactive and automated specification of problems

In algebra systems specification is done by selecting the function (say solve an equation), and supplying the appropriate arguments; the domain is specified by some kind of switch (e.g. real or complex solutions for equations).

In order to allow for interactive specification, we need the underlying knowledge in an explicit form, which can be browsed by the user, and which allows for interactive selection. The universe of this knowledge is organized along three axes (see Fig.1 on p.103), one of which concerns problem-types.

Let us begin with the maximum-example, structured as an (example construction) problem, which may be the result of automated modeling from the hidden formalization, or which may be the result of the user's input: \(^9\)

\[
\begin{align*}
\text{problem ["maximum"]} & \equiv \{ (r, 7) \} \\
I & \equiv \{ (r, 7) \} \\
\eta(r) & \equiv 0 \leq r \\
O & \equiv \{ A, \{ u, v \} \} \\
\rho(u, v, r) & \equiv A = u \cdot v \land (\frac{u}{v})^2 + (\frac{v}{u})^2 = r^2 \land \\
& \quad \forall A' u' v'. A' = u' \cdot v' \land (\frac{u'}{v'})^2 + (\frac{v'}{u'})^2 = r^2 \implies A' \leq A \\
R & \equiv \{ A = u \cdot v, (\frac{u}{v})^2 + (\frac{v}{u})^2 = r^2 \}
\end{align*}
\]

The post-condition \( \rho(a, b, r) \) is, besides \( I \), the characteristic of a problem (type). Unfortunately, the above post-condition is very hard to verify: this could be done for simpler ones like the post-condition for the solution-set \( L \) of an equation \( a = b \), which may be \( \forall \in L \). (\( \lambda v. a \) \( = \lambda v. b \) \( \leq \epsilon \), but not for the maximum-example – we will come back to this issue in section 4. The definition for the notion of a problem is

Definition 3. Given a substitution \( I \) and a set \( O \) of variables with \( (\text{Vars } I) \cap O = \emptyset \), and a set \( R \) of predicates, some predicates \( \eta(\text{Vars } I) \) and \( \rho(\text{Vars } I, \text{Vars } O, \text{Vars } R) \), with \( \rho \) quantifying all free variables by the such-quantifier, then \( L = (I, \eta, O, \rho, R) \) is a problem.

\(^7\) We exclude the important case of substitutions with a value arbitrary but fixed from this paper.

\(^8\) This decision could have alternatives: one could push ahead the modeling to the domain of elementary geometry (comprising formal notions of circle, rectangle, inscribe, etc., or one could employ techniques from artificial intelligence like some geometry tutors.

\(^9\) Note that the intervals like \( 0 \leq \frac{u}{v} \leq r \) from the formalization do not show up in this problem; but it is passed to a subproblem, see the script Maximum Value on p.101.
The elements of $I$ are called **input-items** and those of $O$ are called **output-variables**, $\eta$ is the **pre-condition** and the predicate $\rho$ is the **post-condition**, relating input and output.

$R$ consists of sub-terms of $\rho$, it is redundant for pedagogical reasons.

A problem is **suitable** iff $\eta(I)$ evaluates to true, and a problem is **solved** iff there exists a set $V$ of values for all output-variables, $\overrightarrow{O} = O \times V$ such that $\rho(I, \overrightarrow{O})$ evaluates to true. The set $V$ is called the solution of $L$.

$\eta(I)$ denotes substitution of the $Vars$ $I$ by their respective values. There are restrictions on the (substitution and evaluation) of $\rho(I, \overrightarrow{O})$, the post-condition - see 4.3.

Now, what we need for automated specification is the following: given a formalization, find the appropriate problem-type. For instance, the dozens of problems, found along with the maximum-example in textbooks on calculus, shall belong to one single problem-type.

A preliminary attempt to describe the problem-type, the maximum-example may belong to, is the following:

$$
\begin{align*}
\text{problemtype} & \quad [\text{"maximum"}] \\
I & \equiv \{ \text{fix}_- \} \\
\eta & \equiv \text{map}(\$0 \leq \$) \text{fix}_- \\
O & \equiv \{ m_\_ vs_\_ \} \\
\rho & \equiv \text{let } x_1 = \{ m_\_ \} \cup \{ vs_\_ \} \cup (Vars \, rs_\_); \\
& \quad x_2 = \text{map} \, \text{primed} \, x_1; \\
& \quad \text{in } \text{map} \, (\text{op} \wedge) \, rs_\_ \$ \wedge \$ \\
& \quad \text{forall } x_2 \$: $(\lambda \$ x_1 \$. \$. \text{map} \, (\text{op}) \, rs_\_ \$) \$. x_2 \$ \implies \$. \text{primed} \, m_\_. \$ \leq \$. m_\_. \\
R & \equiv \{ rs_\_. \}
\end{align*}
$$

where $\$ denotes a term-constructor. $\text{primed}$ attaches a $'$ to a variable. $\text{map}$ takes sets instead of lists, and, for instance, creates inequalities for the pairs $\text{fix}_-$. The underscores define identifiers to belong to the meta-language of problem-types (as opposed to the object-language of math).

The design of the language describing how to generate pre- and post-condition from the problem-type is an open question, and the author is not aware of related work. Presently the syntax of $\eta'$ and $\rho'$ (let us call them **p-templates**) is unclear, as well as the details of their generation, i.e. some function $X$

$$
X : P' \times S \longrightarrow P
$$

where $P'$ is a set of p-templates, $S$ is a set of substitutions, and $P$ a set of predicates. This function works for the example as follows

$$
X : (\text{map}(\$0 \leq \$) \text{fix}_- . \{ \text{fix}_- , \{(r,7)\}\}) \longrightarrow 0 \leq r
$$

The function $X$ is necessary to proceed from Def 4 to Def 5.

**Definition 4.** Let $X_1, X_2$ be sets of variables, $P_1$ a set of predicates. Given the sets $I'$, $O'$ and $R'$ of variables, and given two p-templates $\eta'$ and $\rho'$, then $Y = (I', \eta', O', \rho', R')$ is a **problem-type**. $I', O', R'$ are called the **input-components** of $Y$, shortly $IOR'$.

A problem-type is a kind of a general template to be instantiated by a particular formalization:
Definition 5. Let $F_0, F$ be formalizations, $F \equiv (I, O, R)$, $Y \equiv (I', I', O', O', R')$ a problem-type with input-components $IO$ and $IO'$. Let further be $L$ a problem, $P$ a set of predicates, $S$ a set of sets of input-items, and $V$ a set of sets of variables. Then we say $F$ instantiates $Y$ given $F_0$ yielding $L$ iff
\begin{itemize}
  \item[(i)] $F_0 = \emptyset \land$ matching $IO' F$ while generating $\sigma_Y = \text{match } IO' F$
  \item[(ii)] $\exists \eta \in P, I \in S \land \eta = \mathcal{X}(\eta, \sigma_Y)$ and $\eta(I)$ holds
  \item[(iii)] $\exists \rho \in P, \rho = \mathcal{X}(\rho, \sigma_Y)$
  \item[(iv)] $\exists O \in V, L = (I, \eta, O, \rho, R)$
\end{itemize}

where match yields a substitution $\sigma_Y$, and matching is true for $\sigma_Y = \emptyset$. Condition (i) contains a case-distinction concerning whether there is a hidden formalization $F_0$ prepared or not. This is the answer to requirement (1.) on p.92 w.r.t. specification.

With the latter definition we have come to the point, because instantiates can be used to construct a quasi-order, which in turn allows one to construct an acyclic graph.

Definition 6. Given two problem-types $Y_1 = (I_1, \eta_1, O_1, \rho_1, R_1)$ and $Y_2 = (I_2, \eta_2, O_2, \rho_2, R_2)$, and a set $\mathcal{F}$ of formalizations, we say $Y_1$ refines $Y_2$ iff $\forall F \in \mathcal{F}$. $F$ instantiates $Y_2 \Rightarrow F$ instantiates $Y_1$.

The acyclic graph constructed by the quasi-order on problem-types induced by refines, leads to the hierarchy of problem-types, called the problem-tree:

Definition 7. Let $ID$ be a set of strings, and $id_i \in ID$ some elements (for $0 \leq i \leq n, n \leq 2$), called labels, and $\mathcal{Y}$ a set of problem-types with some elements $Y_i \in \mathcal{Y}$. Then we call the acyclic graph ‘problemtree’ with constructor ‘Join’ and nodes in $(ID \times \mathcal{Y})$

datatype probelmtree = Join of ((ID \times \mathcal{Y}) \times (probelmtree list))

a problem-tree iff
\begin{itemize}
  \item[(i)] for all parallel nodes $(id_i, Y_i)$ the labels $id_i$ are pairwise disjoint
  \item[(ii)] $Y_i$ below $Y_j$ iff $Y_i$ refines $Y_j$
  \item[(iii)] $Y_i$ parallel $Y_j$ iff $\neg(Y_i$ refines $Y_j) \land \neg(Y_j$ refines $Y_i)$
\end{itemize}

where below and parallel are relations on the acyclic graph (equivalent to the respective notions on terms). The list of labels $[id_1, \ldots, id_n]$ along the path from the problem-trees root to a problem-type $Y_k$ is called a problem-ID.

Given a problem-tree, we can automatically refine a vaguely formulated problem to a stronger formulated one! Let us look at the part of a problem-tree concerning the maximum-example:

\begin{verbatim}
  : join (("make_fun", Y_k),
     [ join (("by_elimination", Y_k1), []),
       join (("by_new_variable", Y_k2), []), ... ])
  : ...
\end{verbatim}

Given any one of the formalizations of p.95 and the problem-ID ["make_fun"], automated refinement can be done due to the matching of input-items and the evaluation of the pre-condition of $Y_k1$ and $Y_k2$ respectively: If the formalization is $F_I$ or $F_{II}$,
$y_{b1}$ would be chosen, and the method attached would yield $A(a) = 2 a \sqrt{t^2 - (s)^2}$ or $A(b) = 2 b \sqrt{t^2 - (s)^2}$, and if the formalization is $F_{11}$, $y_{a2}$ would yield something like $A(a) = 2 \cdot 7 \sin \alpha \cdot 2 \cdot 7 \cos \alpha$. $y_{k1}$ is addressed by the problem-ID ["makeFun", "by_elimination"] and $y_{k2}$ is addressed by ["makeFun", "by_new_variable"]. The script Maximum_value on p.101 employs the mechanism of refinement.

To the author's best knowledge, the automated refinement on explicit problem-types seems to be a novel mechanism. We eagerly look forward to experiences of how larger portions of knowledge, e.g. all types of equations in the high-school syllabus are to be mapped to a problem-tree, to the respective patterns of input-items and pre-conditions, and how the performance of automated refinement will be.

### 3.3 Solve stepwise by rewriting

A major part of high-school math can be done by rewriting. [Neu01a] gives a survey on the respective topics within the syllabus. Rewriting is very close to what a human mathematician does when applying a certain theorem to a certain formula. Mechanical rewriting, the step by step application of theorems (like $l = r$, as directed rules $l \rightarrow r$) tends to be verbose; [Buc97] showed how to master this verbosity by a 'nested cells representation'.

*Re-engineering the simplifiers* for $\mathcal{N}, \mathcal{Z}, \mathcal{Q}, \mathcal{R}$ and the complex numbers $\mathbb{C}$, all together with the related function constants, is apparently the most straightforward task. Several simplifiers are trivial, such as differentiation or expanding logarithms, and just require basic knowledge on termination and congruence in rewriting.

Many other tasks are not trivial; for instance (a comprehensive survey is given in [Neu01a]), the calculation of the canonical polynomial form over $\mathbb{Z}$ with numerical constants involves conditional rewriting\(^{(10)}\), and thus the basic knowledge as presented e.g. in [NB98] is not sufficient. This is even more the case with rationals or radicals.

Also rewrite orders are of concern, for instance if a special term (say $\sqrt{\cdot}$) needs to be shifted to the root of a term (Knuth-Bendix order !) - see the tactic *Rewrite_Set isolate_root* in the script *square_equation* on p.101.

There is a wealth of very specialized literature on rewriting, but much is 25 years old and thus not easily accessible. In fact, some specific knowledge, buried in any algebra system, may need to be re-invented. But all together, this task is more suited for industrious students of computer mathematics, than for challenging R&D.

*Implementation of 'reverse rewriting'* follows a novel idea, exploiting a suggestion of [Har97] on the combination of algebra systems and theorem provers. The idea is the following:

\(^{(10)}\) While distributing towards polynomial form by $(a + b) \cdot c \rightarrow a \cdot c + b \cdot c$, numeral constants must be contracted by $a \in \mathbb{const}$ and $b \in \mathbb{const} \Rightarrow a \cdot c + b \cdot c \rightarrow (a + b) \cdot c$; both rules together would obstruct termination, if there would not be the condition.
There are topics in high-school mathematics, not really suitable for rewriting, but taught as such. A typical example is factorization while calculating in \( \mathbb{Q}[x] \):

\[
\frac{x - 2}{2x^2 - 2} = \quad \text{Calculate} \quad 8 = 2 \cdot 1^2 \\
= \frac{x - 2}{2x^2 - 2} = \quad \text{Rewrite} \quad 2a - 2b = 2(a - b) \\
= \frac{x - 2}{2(x^2 - 1^2)} = \quad \text{Rewrite} \quad a^2 - b^2 = (a + b)(a - b) \\
= \frac{x - 2}{2(x + 1)(x - 1)} = \quad \text{Rewrite} \quad b = \frac{1}{a} \\
= \frac{1}{2(x + 1)} = \ldots
\]

Indeed, factorizing and canceling in (2)\ldots(4) of the above example are taught as an application of the (inverse) law of distributivity and of laws on binoms, while (1) is skipped as ‘obvious’.

The automated process of ‘reverse rewriting’ is proposed to factorize yielding (4), but without showing the result to the student; rather, certain factors are multiplied again in order to present steps (1)\ldots(3) exactly in this sequence. This is an answer to the requirements (2) and (3) on p.92. The technique of ‘reverse rewriting’ is intended to implement step by step calculation of canonical forms in all domains taught at high-schools.

4 Scripts for reactive user-guidance

4.1 Describing methods

A method of solving a specified and instantiated problem is described by a so-called script. Scripts are formulated in Isabelle/HOL; their syntax, given in Backus normal-form, is

\[
\text{script ::= Script id arg* = body} \\
\text{arg ::= id | ( id :: type )} \\
\text{body ::= expr} \\
\text{expr ::= \% id . expr} \\
\text{let id = expr ( ; id = expr)* in expr} \\
\text{if prop then expr else expr} \\
\text{while prop expr id} \\
\text{repeat expr id} \\
\text{try expr id} \\
\text{(expr @ expr) id} \\
\text{tac ( id | listexpr)* | listexpr | id} \\
\text{type ::= id} \\
\text{tac ::= id}
\]

where \((\cdot)\) belongs to the object-language, id is an identifier with the usual syntax, prop is a proposition constructed by logical operators of Isabelle/HOL, listexpr (called list-expression) is constructed by Isabelle’s list functions like hd, tl, nth, and type are (virtually) all types declared in Isabelle’s version 99. tac stands for tactics, tacticals are written in typewriter font.
% (for λ-abstraction), let ... in and if ... then ... else are already defined in HOL, while checks the given formula, repeat, try, or depend on whether their subexpressions are applicable (see p. 95). Tactics to be done in parallel can be modeled by two tactics, by let ... in or by or.

A script solving the maximum-example is mainly concerned with data-transfer to subproblems, and could look like

```
Script Maximum_value (fixζ : bool list) (μw : real) (ηw : bool list)
    (υw : real) (πυw : real set) (σw : bool) =
        let
            ζw = (hd a (filter (Testvar ζw)) ηw);
            ηw = (if #1 < Length ηw
                    then (Subproblem (Reals, ["make_fan"]:no; met) [μw, ζw, ηw])
                    else (hd ηw));
            μw = Subproblem (Reals,["function", "max_of", "on_interval"],
                             maximum_on_interval) [ ζw ζw ηw ]
        in (Subproblem (Reals,["tool", "find_values"]:find_values)
             [ μw (Dths ζw), ζw μw (dropWhile (ident ζw), ηw)])

Testvar filters that relation, which contains the variable describing the value to be maximized. The tactic Subproblem (Reals,[make_fan]:no; met) [μw, ζw, ηw] deserves special attention: it takes the arguments domain Reals, problem-type [make_fan] and the formalization [μw, ζw, ηw], but it does not specify a method (no; met). This can be done because of the mechanism of refinement, Def 6: dependent on the modeling of the root-problem, one of the formalizations is passed in by the formal parameters of the script; when calling the subproblem, the problem-type is refined to ["make_fan", "by_elimination"] or ["make_fan", "by_new_variable"]. This mechanism definitely generalizes the call of subproblems in comparison to function calls in other program languages.

Another script, typical for rewriting, describes a method for solving an equation containing square-roots, where the equation can be solved by isolating the roots and squaring the whole equation, is as follows

```
Script square_equation (eqw : bool) (υw : real) (σw : real) =
    let εw =
        (while (not o is; root_free)
            εw (let
                εw = try (Rewrite_Set simplify False) eqw;
                εw = try (repeat (Rewrite assoc_plus_inv False)) εw;
                εw = try (repeat (Rewrite assoc_plus_inv False)) εw;
                εw = try (Rewrite_Set isolate_root False) εw;
                in ((Rewrite square_equation_left True) or
                    (Rewrite square_equation_right True)) εw)
            eqw;)
        εw = try (Rewrite_Set Inst [bdw, ζw] norm_equation False) εw;
        Lw = Subproblem (Reals, ["equation", "univariate"], no; met) [ εw ζw σw ]
        in Check_elementwise Lw Assumptions)
```

where Rewrite has a boolean argument pushing the assumptions of conditional rewrite-rules into the global assumptions or not.
4.2 Reactive user-guidance

What we call 'reactive user-guidance' here is a consequence of the math-engine's design: the math-engine is capable of automated modeling (given a formalization) and of automated specification and refinement, as already discussed. The user, on the other hand, is free to supply a step of his or her own choice, and the math-engine will give feedback. The feedback during modeling concerns unknown or missing items and the suitability of problem-types during specification. If the user gets stuck, the math-engine is able (in principle) to supply the next step – in reaction to previous steps of the user.

This concept is easy to implement for modeling and specifying, but it had turned out to be hard for solving (which is guided by scripts). At each tactic, Subproblem, Rewrite_Set_Inst, Rewrite_Set, Rewrite etc., control is passed to the user. The interpreter of the scripts is concerned with the following tasks:

1. beginning with the last tactic done (or the root of the scripts body) find the next tactic to do; this may fail due to a 'misleading' tactic
2. present this tactic, i.e. the user-tactic associated to the script-tactic found, to the student as a suggestion for the next step
3. receive the student's input (assumed to be a tactic here for reasons of simplicity, and not an expression; a tactic, however, which may be different from the one suggested by the script) and check if it is applicable at the present proof-state; if so, apply and promote the proof-state, otherwise notify the student
4. locate the tactic associated with the input tactic, which may be classified as misleading continue with 1.

This kind of interpretation (the details of which belong to the domain of compiler construction, see [Neu01a] and which exceed space limits) is a major advance towards a flexible dialog, where the system and the user are partners on an equal basis: the user can be active and propose tactics, and the system checks them for applicability at the present proof-state, or the user passes activity to the system and requests a proposal for the next step.

The reader may have noted that the scripts do not contain any hint for the dialogue; this is done in a separated module.

4.3 On the correctness of scripts

The application of a method, i.e. the evaluation of a script, is guarded by a specification of the problem containing a post-condition. It has been mentioned on p. 96, that some postconditions cannot easily be verified for a particular example, e.g. for the example

In this case it would be an interesting challenge for automated deduction to verify the correctness of the script w.r.t. the post-condition. It is not the example constructed, which is of concern, but rather the algorithm constructing such examples is of concern (one level of abstraction higher)!

Such a proof of correctness would involve the semantics of scripts, which are formulated in Isabelle/HOL, i.e. meta-language and object-language could be within the same logic. [Nip98] could be a model for this task. This proof would also concern the semantics of tactics involving the proof-tree and its branch-types; this is an even harder task. Altogether, it's an issue, but not considered urgent for the practical usage of the prototype.
5 Explanations by reflection

Reflection is a term, which recently has become folklore with the program-language Java; it denotes facilities of the software-system to inspect its own language-constructs. Reflection in context with explanations means, that requests for explanations should not somehow be foreseen by an author of the system; but rather the system should just reflect its state and its knowledge and only on user request will the system exhibit its knowledge and mechanisms in reasonable portions and steps towards more and more details.

5.1 A separated layer of explicit knowledge

Math knowledge is separated from the math-engine, which interprets the knowledge. Due to verbal advice from the creator of Theorema©[BJ98] the knowledge is organized along three axes (see Fig.1): the axis of problem-types as presented in 3.2, the axis of domains from 4.1, and the axis of domains, which has already be given by Isabelle's hierarchy of theories [Pau94].

The user will access this knowledge in leisure browsing, but also with particular requests arising from special situations. This raises issues related to the high structure of the knowledge; the details are not yet designed.
5.2 The representation of the proof-state

is done on a work-sheet, one of which can be found on p.94. A work-sheet produced by
the script `square_equation` on p.101 could look like

\[ L = \text{solve_root_equ} \left( \sqrt{9 + 4x} = \sqrt{\sqrt{5} + x} \right) \ (bdv = x) \ (e = 0) \]

1. \[ \sqrt{9 + 4x} = \sqrt{\sqrt{5} + x} \]
   
   \text{Rewrite (} \text{square_equation_left}, \ a \geq 0 \ \& \ b \geq 0 \ \Rightarrow (a = b) = (a^2 = b^2) \)

1.1. \[ (\sqrt{9 + 4x})^2 = (\sqrt{\sqrt{5} + x})^2 \]
   
   \text{Rewrite Set simplify}

1.2. \[ 9 + 4x = 5 + 2x + 2\sqrt{3x + x^2} \]
   
   \text{Rewrite Set isolate_root}

1.3. \[ \sqrt{5x + x^2} = \frac{(9 + 4x)}{(5 + 2x)} \]
   
   \text{Rewrite (} \text{square_equation_left}, \ a \geq 0 \ \& \ b \geq 0 \ \Rightarrow (a = b) = (a^2 = b^2) \)

1.4. \[ (\sqrt{5x + x^2})^2 = \left(\frac{(9 + 4x)}{(5 + 2x)}\right)^2 \]
   
   \text{Rewrite Set simplify}

1.5. \[ 5x + x^2 = 4 + \frac{1}{2}x + x^2 \]
   
   \text{Rewrite Set Inst [(bdv,x)] normalize_equation}

2. \[ x - 4 = 0 \]
   
   \text{Subproblem (} \text{R_equation, univar}, \ e) \]

3. \[ L_1 = \text{solve_univar} (x - 4 = 0) \ (bdv = x) \]
   
   \text{Apply Method (} \text{R_solve_linear} \)

3'. \[ L_1 = \{4\} \]

\[ \text{Check elementwise} \ 0 \leq \sqrt{x} + \sqrt{\sqrt{5} + x} \ \& \ 0 \leq 9 + 4x \ \& \ 0 \leq 2 + \frac{5}{x} \ \& \ 0 \leq 2 + x \]

\[ L = \{4\} \]

Reflecting the proof-state, such a work-sheet cannot be edited arbitrarily by the
user; there are various solutions possible to this requirement, see for instance [Asp00].
Another model is [Wen99].

For the user-group envisaged, i.e. high-school students, the work-sheet should look
like a calculation done by paper and pencil. On the other hand, many technical details
need to be displayed, as soon as the user starts to inquire. In the example above, for
instance, at line (1.) and (1.3.) the applied theorem contains a condition, which needs
to be instantiated and stored with the assumptions of this (sub-)problem. How should
this be displayed? Many similar design decisions have still to be made!

6 Conclusions

The 'Calculator-approach', attempted in this paper for the design and the development
of educational software for mathematics, lead to concepts and a prototype, very
different from existing algebra systems and presumably simpler to use than deduction
systems. This prototype demonstrates the feasibility of the concepts concerning
interactivity, and it is used to make teachers aware of what they can request from
state-of-the-art math tools [Neu01b].

6.1 Conceptual achievements

Conceptual achievements, established by merging techniques from algebra systems and
from deductive systems and shown as feasible by the prototype, are the following.

- A proof-state justifying each step as correct or incorrect (i.e. checking a tactic
  applicable or not applicable w.r.t. the current proof-state, see 2.2), or eventually
  misleading

\text{11} The theorems use the syntax of Isabelle/HOL, which uses = in a strange looking
way: \((a = b) = (a^2 = b^2)\) etc.
- The knowledge in a separate language layer, i.e. the 3D-universe of math in human readable format (5.1); this can be inspected by the user and interpreted by the math-engine as well
- Interactive and automated specification of a domain (i.e. an Isabelle theory), a problem (type; see sect.3.2), and a method
- Step-by-step execution of methods, given by scripts whose interpreter passes control to the user at each tactic and resumes guidance (4.2)
- The prototype’s math-engine is ready for implementing the math-knowledge (about 70% of the syllabi are appropriate).
- The math-engine as implemented is an appropriate basis for establishing a novel kind of dialog, where the system and the user interact as partners on an equal basis.

6.2 Open questions (?), and future work (!)

- Formal underpinning of the semantics of tactics (2.2) and tacticals (4.1), in order to prove a script w.r.t. the postcondition of the respective problem ?
- Completeness of the set of tactics (2.2), and their usability for students (to be investigated in field-studies) ?
- Problem-hierarchy appropriate to incorporate all math-knowledge for high-school and to provide for adequately efficient problem refinement ?
- Re-engineering of the basic functions of algebra systems (solve, simplify, differentiate etc.) and implementation along the respective axes in the 3D-universe (5.1) !
- Enhanced interactivity in the prototype: dynamic search in the 3D-universe, go back to previous proof-states, high-level description of dialogs and user-model !

The approval, which the prototype has already gained from teachers, seems to make worth the effort to continue clarification of theoretical foundations, to implement the math knowledge according to the syllabus, and to extend the prototypes functionality.

References


Computer Algebra meets Automated Theorem Proving: A Maple-PVS Interface

Martin Dunstan\textsuperscript{1}, Hanne Gottliebsen\textsuperscript{2}, Tom Kelsey\textsuperscript{2}, and Ursala Martin\textsuperscript{2}

\textsuperscript{1} NAG LTD, Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, UK
\textsuperscript{2} School of Computer Science, University of St Andrews, Fife KY16 9SS, UK

Abstract We present an interface between the automated theorem prover PVS and the computer algebra system Maple. The interface is designed to extend Maple with formal proof capabilities. In addition to proofs using standard PVS libraries, we can prove properties in the theory of real analysis using a new PVS library which has been developed for this specific purpose. We demonstrate applications of the interface in topics such as algebraic equality, definedness with respect to parameters, and ordinary differential equations.

1 Introduction

The objective of the project described in this paper is to develop a specific interface which links the major commercial computer algebra system\textsuperscript{1} Maple\textsuperscript{18} to the widely used automated theorem prover\textsuperscript{2} PVS\textsuperscript{19} in a way which minimises the inherent problems of linking two unrelated systems. We also utilise recent extensions to PVS\textsuperscript{10, 3} which allow reasoning in the theory of real analysis, so that properties of symbolic expressions can be described and checked. The project is in approach similar to the PROSPER toolkit\textsuperscript{11}, which allows systems designers using CAD and CASE tools access to mechanised verification. The PROSPER paradigm involves the CAD/CASE system as the master, with a slave proof engine running in the background. We have also adopted the PROSPER architectural format: a core proof engine which is integrated with a non-formal system via a script interface. Our target, however, is the engineering/scientific/mathematical community of CAS users. These users interact with Maple, making calls to PVS, which acts as a black box for the provision of formal proofs.

The aims of the paper are

1.1 Previous and related work

Previous approaches to the integration of CAS and ATP are discussed and classified in\textsuperscript{9}. Proposed solutions of the common problem of communicating mathematical information between the CAS and ATP systems fall into three categories: the common knowledge, sub-package and specific interface approaches.

\textsuperscript{1} henceforth CAS
\textsuperscript{2} henceforth ATP
The common knowledge approach It is possible to define standards for mathematical information which can, in principle, be understood by any mathematical computation system. Examples include the OpenMath project [1], and protocols for the exchange of information between generic CAS and ATP systems [6, 14]. This approach is the most attractive, since it maximises the potential for large-scale, open-source developments of combined ATP and CAS systems. However, success depends on wholesale acceptance of the standards by both the CAS and ATP communities. Since there are currently many unresolved technical, legal, cultural and semantic issues, we concentrate on the remaining two approaches.

The sub-package approach Here communication issues are side-stepped by building an ATP within a CAS. Examples include Analytica [5], REDLOG [12], the Theorema project [7, 8], and a logical extension to the type system of the Axiom CAS [17, 20]. The approach has the advantage that communication is easy, and the disadvantage that implementation of an ATP in a language designed for symbolic/numeric computation can be hard. In particular, there may be problems with misleading or incorrect output from CAS routines, which adversely affect the soundness of the proof system. For example, the simplification error described in Section 3.1 could lead to an undetected division by zero, which could propagate logical errors.

The specific interface approach The third common approach involves the choice of preferred CAS and ATP environments, and the construction of a specific interface for communication between them. Examples include Maple-HOL [15], Maple-Isabelle [4], and Weyl-NuPrI [16]. The approach has the advantage that the target CAS and ATP are implemented and developed by experts in the respective technologies, and the disadvantage that communication problems remain. Another issue is the relationship between the systems. In the examples mentioned above, the primary system in use was the ATP, and the CAS was used as an oracle to provide calculation steps. By contrast the motivation for our work is to support the users of computer algebra systems in their work by giving them the opportunity to use the rigour of theorem prover when they wish, completely automatically in some cases. This has the advantage that users can use all the facilities of the CAS, but the theorem prover implementation can be restricted. Since Maple is a programming language, calls to a prover can be embedded in procedures and made invisible to the user.

1.2 Maple

Maple [10] is a commercial CAS, consisting of a kernel library of numeric, symbolic and graphics routines, together with packages aimed at specific areas such as linear algebra and differential equations.

The key feature of version 6 of Maple is that it was designed to run a subset of the NAG numerics library. We utilise this ability to extend Maple by running PVS as a subprocess.

1.3 PVS

PVS [19] supports formal specification and verification and consists of a specification language, various predefined theories and a theorem prover which supports a high level
of automation. The specification language is based on classical, typed higher-order logic and supports predicate and dependent sub-typing.

PVS is normally run using GuM or Xemacs for its interface, but can also be run in batch mode or from a terminal interface. However, running PVS via emacs provides many useful features such as abbreviations for prover commands and graphical representations of proofs and library structures. The core of PVS is implemented in Allegro Common Lisp.

1.4 The PVS Real Analysis Library

Our goal is to provide automatic support for reasoning about real valued CAS expressions. The basic PVS libraries are insufficient for proving properties such as continuity or convergence of functions. It was therefore necessary to extend PVS with (i) libraries of real analysis definitions and lemmas, and (ii) specialist proof strategies.

A description of a PVS library of transcendental functions was provided in [13]. The library can be used to check continuity of functions such as

\[ e^{x^2 + |1-x|} \]  

Further illustrative examples are given in [2].

Since [13] a convergence checker has been added to the library. This can be used to check whether a certain function has a limit at some point (or indeed everywhere in its domain). We can prove, for example, that the function

\[ -\pi - 1 + \pi \star e^{1-\cos(x)} \]  

has a limit at the point

\[ x = \arccos(1 - \log(\frac{1 + \pi}{\pi})) \]  

The convergence checker is implemented in the strategy conv-check, and it works in the same syntax-directed way as the continuity checker, and so has similar capabilities and limitations.

2 Implementation

In this section we describe the work undertaken to develop our Maple-PVS system. We have created a tightly coupled system under UNIX with PVS being controlled by Maple as if it was a "normal" user. That is, PVS believes that it is interacting with someone entering commands from a terminal rather than, for example, a pipe to another program.

2.1 Design Issues

The overall view of our system is one in which the user interacts with the CAS, posing questions and performing computations. Some of these computations may require ATP technology, either to obtain a solution or to validate answers obtained by existing computer algebra algorithms. In such cases the CAS will present the theorem prover with a number of lemmas and request it to make some proof attempts. The results of
these attempts, whether successful or otherwise, will guide the rest of the computation within the CAS: the theorem prover acts as a slave to the CAS.

The system has a tightly coupled architecture. In such a system, the theorem prover shares the same resources as the computer algebra system and is directly controlled by it.

![Tightly Coupled System Diagram](image)

**Figure 1: Tightly Coupled System**

### 2.2 Extending Maple

Although Maple provides its own programming language, it was necessary to make use of the Maple interface to external functions written in C. Technical details are given in [2]. These C functions provide a basic low-level interface to PVS. Built on top of these are a number of higher-level functions written in the Maple language. For example, the Maple function used to start a PVS session with a particular context is:

```maple
PvsStart := proc(dir::string)
    # Start PVS using imported C function
    pvs := PvsBegin();

    # Send data to PVS via imported C function
    PvsSend("(change-context \"" || dir || \"")");

    # Wait for prompt using another imported C function
    PvsWaitForPrompt(pvs);

    pvs;
end proc
```

These Maple procedures are supplied as a module, which allows users to import them as a library package. The library must be added to Maple by each user before the interface can be used. The library can then be accessed at each session by the command:

```maple
> with(PVS);

[PsLineFind, PvsPrintLines, PsProw, PsQEDfind, PvsReadLines, PvsSendAndWait, PvsStart]
```
2.3 Maple-PVS Communication

Maple-PVS communication has been implemented in two parts. Firstly, a simple lexical analyser (pvs-filter) recognises PVS output and translates it into a format that is easier to parse by other tools (such as the Maple interface). Secondly, a small Tcl/Tk application (pvs-ctl) to act as a broker between Maple, PVS and the Maple-PVS user. This application is launched from Maple as the child process instead of the PVS-Allegro LISP image and uses pvs-filter to simplify the processing of PVS-Allegro output.

![Figure 2: Maple-PVS System](image)

Under normal use, Maple sends PVS commands to pvs-ctl which passes them directly to PVS-Allegro. Responses are translated by pvs-filter and examined by pvs-ctl. Anything that needs the user’s attention is handled interactively by pvs-ctl allowing the user to respond directly to PVS-Allegro without Maple being aware of it. Status messages from PVS-Allegro are displayed graphically by pvs-ctl along with other output describing how proof attempts are progressing. Again, none of this reaches Maple—only the information that Maple actually needs. At present, Maple only needs to know about PVS prompts (so that it can send the next command), and Q.E.D. messages indicating that a proof attempt was successful.

The benefits of this system are significant: for the Maple programmer PVS appears as a black-box which is controlled via a trivial protocol. Maple sends a command to the black-box and reads response lines until a prompt is found. If any of these lines contains a Q.E.D. line then the command was successful, otherwise it was not. Simplicity is important because Maple is designed for solving computer algebra problems, and not for text processing.

For the Maple-PVS user the pvs-ctl application provides a graphical display of the current state of the system. Not only can progress be monitored, but interaction with PVS-Allegro is available when needed. If PVS stops in the middle of a proof attempt and asks for a new rule to apply, the novice user can respond with (quit) while the expert user might be able to guide the proof further.

2.4 A Simple Example

We now provide a straightforward example of the use of the interface. We assume that the Maple user has installed locally the C code, shell scripts and Maple library described in Section 2, and PVS. The first task is to initialise the interface, and check that we can prove that $2 + 2 = 4$. 
> \texttt{pws := PvsStart("./pvslib");}

The \texttt{PvsStart} command launches a \texttt{Tcl/Tk} window and opens communications with a PVS session using libraries found in the given directory.

> \texttt{ex1 := PvsProve(pws, "g: FORMULA 2 + 2 = 4", ", ");}

The \texttt{PvsProve} command takes (i) a PVS session identifier, (ii) a formula in PVS syntax, (iii) a PVS library - the default is the prelude, and (iv) a PVS proof command - the default is \texttt{ASSERT}. The result of this command is shown in Figure 3.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{maple-pvs-interface.png}
\caption{Tcl/Tk window for the Maple-PVS interface}
\end{figure}

We confirm in Maple that the proof was successful using the \texttt{PvsQEDfind} command:

> \texttt{PvsQEDfind(ex1);}

\begin{verbatim}
true
\end{verbatim}

The above example shows that the Maple user controls the interface using Maple commands in a Maple session. The user can check that proof attempts have succeeded without needing to interact with (or even view) the Tcl/Tk window. This is present only as a gateway to PVS, used when proof attempts fail, or when a record of the logical steps used in a proof is needed.

3 Basic Applications

In this section we demonstrate the use of the Maple-PVS interface to obtain proofs which use lemmas from the real analysis library discussed in Section 1.4.
3.1 Algebraic Equality

Consider the expressions \( a = |x - y| \) and \( b = |\sqrt{x} + \sqrt{y}| |\sqrt{x} - \sqrt{y}| \), where \( x \) and \( y \) are positive reals. Suppose a Maple user wishes to verify that \( a = b \). The commands available are

\[
\begin{align*}
& a := \text{abs}(x-y); \\
& \quad a := |x - y| \\
& b := \text{abs}(\text{sqrt}(x)+\text{sqrt}(y)) \ast \text{abs}(\text{sqrt}(x)-\text{sqrt}(y)); \\
& \quad b := |\sqrt{x} + \sqrt{y}| |\sqrt{x} - \sqrt{y}| \\
& \text{assume}(x > 0); \text{assume}(y > 0); \\
& > \text{verify}(a,b,\text{equal}); \\
& \text{FAIL}
\end{align*}
\]

It is evident that there is a problem with the robustness of the Maple \texttt{verify} command, which is equivalent to the test \texttt{signum}(0, a-b, 0) = 0, but which fails to evaluate \( a-b \) to zero. The proof of this identity using the Maple-PVS interface is obtained with the command:

\[
\begin{align*}
& \text{ex3 := Prove}(\text{pvs}, \text{"sqrt_eq: LEMMA FORALL (x,y:posreal) : \text{abs}(\text{root}(x,2)-\text{root}(y,2)) \ast \text{abs}(\text{root}(x,2)+\text{root}(y,2)) = \text{abs}(x-y)"}, \text{"roots"}, \text{"then (skolem!)(use "sqrt_difference")(use "abs_mult")(assert")});
\end{align*}
\]

This example demonstrates the greater proof power of the interface. Maple alone fails to obtain the required identity. Unfortunately, we need several explicit commands to guide the PVS proof.

**Definedness and Parameters** Consider the definite integral

\[
\int_{a}^{b} \frac{1}{x-a} \, dx
\]

where \( a \) and \( b \) are positive real parameters. Maple returns the solution \( \log(b-a) - \log(a) - i\pi \), which, when \( a = 1 \) and \( b = 2 \), reduces to \(-i\pi \) which is a complex number, and hence incorrect. Maple does not check that the function is defined everywhere in the interval of integration; in this case that \( a \) is not in \((0, b)\).

We use the interface to prove that the integrand is not defined when \( a \in (0, b) \) via the command:

\[
\begin{align*}
& \text{Prove}(\text{pvs}, \text{"def: LEMMA FORALL (a,b:posreal) : \text{b > a IMPLIES EXISTS (x:real) : 0 <= x AND x <= b AND not(member[real](x,([z:real|z /=a]))))",} \\
& \text{"", \"then (skosimp)(inst 1 "a!1") (grind")});
\end{align*}
\]

For this example the proof argument to \texttt{Prove} also requires several commands which induce a sequential PVS proof by repeated Skolemisation and flattening, explicit instantiation, and use of the \texttt{grind} tactic.

The next stage of the development of the interface is the provision of a suite of strategies for performing analysis proofs with much less user guidance.
3.2 Using the Real Analysis Library

By using the real analysis library described in Section 1.4, we can prove continuity and convergence directly for a wide range of functions. To illustrate this we use the PvsProve command with top_analysis as the library, and with either cts or conv-check as the proof strategy. For example, we prove that

\[ f(x, y) = \frac{1}{e^{\pi - |\cos(x)|}} \]  

is defined and continuous on the real number line using the cts strategy:

\[ \text{ex5 := PvsProve(pvs, "g: LEMMA FORALL (y:real) :} \]
\[ \text{continuous(lambda (x:real) :} 1/\text{exp}(\pi - \text{abs}(6*\text{cos}(x))),y"),} \]
\[ \text{"top_analysis", "cts"};} \]

We can also prove that

\[ e^{1-\cos(x)} - \pi - 1 \]  

is convergent (i.e. has a limit) at

\[ x = \arccos(1 + \log(\frac{1 + \pi}{\pi})) \]

via the conv-check strategy:

\[ \text{ex6 := PvsProve(pvs, "g: LEMMA convergent(LAMBDA (x:real) :} \]
\[ \text{pi-1+pi*exp(1-\cos(x)),acos(1+ln((1+pi)/pi)))"},} \]
\[ \text{"top_analysis", "conv-check"};} \]

We can also verify that expressions such as

\[ -\pi e^{1-\cos(x)} \]

can never be zero, using the grind strategy:

\[ \text{ex1 := PvsProve(pvs, "g: FORMULA FORALL (x:real) :} \]
\[ \text{-pi*exp(-\cos(x)+1)/= 0", "top_analysis", "grind :defs NIL"};} \]

These examples demonstrate the inference capability and expressivity of the interface augmented with a library of analytic proof strategies. The results cannot be proved within Maple, and are not easy to prove by hand.

4 Generic Examples

In the previous Section we demonstrated the basic use of the interface: the user identifies a property to be checked, and enters the relevant code into Maple-PVS procedure calls. This methodology is basically a streamlined version of the user having Maple and PVS acting separately, with the user deciding upon an explicit proof/computation strategy. The next stage is to build interface commands into Maple procedures, so that formal proof happens automatically, and the results of the proofs are used in generic procedure output.
4.1 Continuity Checking

Maple has an inbuilt procedure, iscont, for checking the continuity of functions over an interval. The procedure returns true, false or FAIL, since continuity is undecidable in general. Unfortunately the procedure can return incorrect results even for input that, by inspection, is continuous over \( \mathbb{R} \):

\[
\begin{align*}
> & \ f := 1/(\cos(x) + 2); \\
& \ f := \frac{1}{\cos(x) + 2} \\
> & \ \text{iscont}(f, x=-\text{infinity}..\text{infinity}); \\
& \ false
\end{align*}
\]

Our approach is to write a procedure (Figure 4) which takes a real valued function and a range, and returns true, false or FAIL depending on the result of proofs in PVS using the real analysis library:

\[PVS\text{iscont} := \text{proc}(f::\text{algebraic}, \mathfrak{u}::\text{real}, \mathfrak{v}::\text{real})
\]
\[
\text{local} \ \mathfrak{wus}, \ \mathfrak{wusf}; \\
\ \mathfrak{wus} := \text{PvsStart( "./pvslib")}; \\
\ \mathfrak{wusf} := \text{maple2pvs}(f); \\
\ \text{if} \ \text{PvsQEDfind(PvsProve(\mathfrak{wus},}
\ \text{"g: LEMMA FORALL (x: \mathfrak{u}[\mathfrak{v}, \mathfrak{w}]): continuous(\mathfrak{wusf}, x)",
\ \text{"top\_analysis", "cts")})
\ \text{then return true}
\ \text{else}
\ \ \text{if} \ \text{PvsQEDfind(PvsProve(\mathfrak{wus},}
\ \text{"g: LEMMA EXISTS (x: \mathfrak{u}[\mathfrak{v}, \mathfrak{w}]): discontinuous(\mathfrak{wusf}, x)",
\ \text{"top\_analysis", "discts")})
\ \ \text{then return false}
\ \ \text{else return "FAIL"}
\ \end{if}
\ \end{if}
\end{proc}
\]

Figure 4: PVSiscont: using the Maple-PVS interface to check continuity.

Applying the procedure to the above example gives:

\[
\begin{align*}
> & \ \text{PVSiscont}(1/(\cos(x) + 2), -\text{infinity}, \text{infinity}); \\
& \ true
\end{align*}
\]

This example demonstrates the use of the interface to improve the performance of Maple procedures which check analytic properties of functions.

4.2 A Generic Application to IVPs

Consider the initial value problem (IVP)

\[
y'(x) = f(x, y), \ y(a) = \eta
\] (8)
where \( y' \) denotes the derivative of \( y(x) \) with respect to \( x \). Let \( D \) denote the region \( a \leq x \leq b \) and \(-\infty < y < \infty \). Then Equation 8 has a unique differentiable solution, \( y(x) \), if \( f(x, y) \) is defined and continuous for all \( (x, y) \in D \), and there exists a positive constant \( L \) such that

\[
|f(x, y_0) - f(x, y_1)| \leq L|y_0 - y_1|
\]

holds for every \((x, y_0)\) and \((x, y_1)\) \( \in D \).

Our intention is to use the interface and the real analysis library to verify conditions on the input function \( f(x, y) \), and the Maple solution, \( y(x) \), of the IVP. We now describe a methodology for validating and improving Maple procedures for solving IVPs of the form

\[
y'(x) = r(x) - q(x)y(x), \quad y(a) = \eta, \quad x \in [a, b]
\]

We can use the interface to check the following requirements on inputs (bearing in mind that each input can be a complicated symbolic expression involving parameters):

1. \( r(x) \) and \( q(x) \) are continuous over \([a, b]\);
2. \( r(x) - q(x)y(x) \) is continuous, Lipschitz, and/or differentiable over \([a, b]\).

Answers to these questions provide a formal check on the existence and uniqueness of solutions for the given finite range. For example, we proved the continuity of \( \sqrt{x} \) for all positive \( x \) in Section 3.1. We can prove that \( \sqrt{x} \) is not Lipschitz for \( x \in (0, 1) \) using the interface. Information regarding existence and uniqueness can be used to fine tune the procedure used to obtain a solution, by using relevant \texttt{assumes} clauses in Maple (e.g. \texttt{assume(r(x) - q(x)y(x), continuous)} so that specialised solution techniques can be safely used.

Once a solution, \( y(x) \), has been obtained, we can use the interface to check properties such as

1. \( y'(x) - f(x, y) = 0 \) - the solution satisfies the problem;
2. \( y(a) = \eta \) - the initial value is preserved;
3. \( y(x) \) has removable poles, non-removable branch points and/or is itself continuous.

The prototype Maple procedure shown in Figure 5 takes \( r(x), q(x), a, \eta \) and \( b \), and supplies answers to some of the above questions using the internal Maple \texttt{dsolve} procedure for obtaining \( y(x) \).

Maple does have built-in procedures for answering some of these questions, but, as shown in Section 3.1, can fail to detect the equality of two straightforward expressions and the continuity of a simple function. Using the interface helps the user to validate both the input and output of problems, and hence leads to improved use and understanding of the CAS.

5 Conclusions

We have presented an interface between Maple and PVS. The interface gives Maple users access to the proof capabilities of PVS, thus providing a means to gain more formal knowledge of Maple results.

We have created a tightly coupled interface between Maple and PVS, under which PVS is controlled by Maple. A small Tcl/Tk application sits between Maple and PVS, so that PVS looks like a black box to the Maple user. However, it also allows a more experienced user to interact directly with PVS.
\[
qsolve := \text{proc}(r, q, a, \eta, b) \\
\text{local } psv, z1, z2, z3, z4, z5, sol, \text{diffsol}; \\
psv := \text{PvsStart( "/pvslib")}; \\
z1 := \text{PvsProve}(psv, \\
\quad \text{"g: FORMULA FORALL } (v:I[a,b]) : \text{continuous}(\lambda (x:I[a,b]) : r(x), v"), \\
\quad \text{"top\_analysis", \"cts\"}); \\
z2 := \text{PvsProve}(psv, \\
\quad \text{"g: FORMULA FORALL } (v:I[a,b]) : \text{continuous}(\lambda (x:I[a,b]) : q(x), v"), \\
\quad \text{"top\_analysis", \"cts\"}); \\
\text{if not (PvsQEDfind(z1) and PvsQEDfind(z2)) then \text{ERROR(\"invalid input\")} else \\
sol := \text{dsolve}(\{ \text{diff}(y(x), x) = r(x) - q(x)y(x), y(a) = \eta, y(x) \}); \\
\text{diffsol} := \text{diff}(\text{sol}, x); \\
z3 := \text{PvsProve}(psv, \\
\quad \text{"g: FORMULA FORALL } (v:I[a,b]) : \text{diffsol}(v) = r(v) - q(v)\text{sol}(v"), \\
\quad \text{"top\_analysis", \"grind\"}); \\
z4 := \text{PvsProve}(psv, \text{"g: FORMULA sol(a) = eta", \"top\_analysis", \"grind\"}); \\
z5 := \text{PvsProve}(psv, \\
\quad \text{"g: FORMULA FORALL } (v:I[a,b]) : \text{continuous}(\lambda (x:I[a,b]) : \text{sol}(x,v"), \\
\quad \text{"top\_analysis", \"cts\"}) \\
\text{fi; \\
\text{if not (PvsQEDfind(z3) and PvsQEDfind(z4) and PvsQEDfind(z5)) then \\
\text{ERROR(\"invalid solution\")} else sol \\
\text{fi; \\
\text{end}} \\
\]

Figure 5: \texttt{qsolve}: formal checks on ODE input/output

In Section 3.1 we saw that Maple can fail to recognise a seemingly obvious equality, which could lead to an undetected division by zero, and also that Maple might apply standard procedures without checking validity of the input. In Section 3 we showed how the interface can be used to correct these errors.

Our aim is to extend the applicability of the interface in two ways. Firstly the extension of the real analysis library discussed in Section 1.4 by adding new strategies, and secondly by providing Maple procedures which automate the checking of validity of input and output, as described in Section 4.2. These extensions will require an improvement in the communication between the two systems, both in terms of syntax of expressions, and in decision procedures based on failed proof attempts.

References


Equality in Computer Algebra and Beyond

James H. Davenport*

Department of Mathematical Sciences, University of Bath, Bath BA2 7AY, England
J.H.Davenport@bath.ac.uk

Abstract Equality is such a fundamental concept in mathematics that, in fact, we seldom explore it in detail. As is often the case, the computerisation of mathematical computation in computer algebra systems on the one hand, and mathematical reasoning in theorem provers on the other hand, forces us to explore the issue of equality in greater detail.

In practice, there are also several ambiguities in the definition of equality. For example, we refer to \( \mathbb{Q}(x) \) as "rational functions", even though \( \frac{x}{x-1} \) and \( x+1 \) are not equal as functions from \( \mathbb{R} \) to \( \mathbb{R} \), since the former is not defined at \( x = 1 \), even though they are equal as elements of \( \mathbb{Q}(x) \).

The aim of this paper is to point out some of the problems, both with mathematical equality and with data structure equality, and to explain how necessary it is to keep a clear distinction between the two.

1 Introduction

Equality is fundamental in mathematics. As we come to encode mathematical computation in computer algebra systems, we have to define, sooner or later, precisely what we mean by equality. Although computer algebra dates back to 1953, the first real discussion of the meaning of equality seems to have been published in 1969 [6].

In general, abstract mathematical structures such as "Ring" are defined with certain algebraic operations (+, −, 0, *, 1 in the case) but without equality. The corresponding constructive counterparts (categories in the Axiom [28] sense; magmas in the Magma [4, 5] sense) tend to be defined with, in addition, an explicit equality operator.

One reason for this is that equality appears even when one does not expect it. For example, the definition of an \( n \times n \) determinant makes no appeal to the concept of equality; however, the fast (e.g. Gaussian elimination) algorithms to calculate it (\( O(n^3) \) or better) do need equality, and if we don’t have equality, we are forced to an \( O(n!) \) algorithm.

Another problem arises when we build algebraic structures hierarchically [12]. Every field is actually a greatest common divisor domain, so we need to implement a \( \text{gcd} \) function. This is apparently trivial, since \( \text{gcd}(x, y) = 1 \) in a field, unless \( x = y = 0 \), in which case the g.c.d. is also 0. This means that structures that do not support equality (such as those defined in section 2.4) cannot be treated as fields in the Axiom sense.

* This is an expanded version of one section of [10], and the author is grateful to the participants at Mathematical Foundations of Computer Science for their interaction.
2 Equality in Computer Algebra Systems

All computer algebra systems support some notion of equality. We expect to be able to type

\[(x+1)^2 \equiv x^2 + 2x + 1\]

and be told that this is true, even though, as strings or even parse trees, they are blatantly unequal. Indeed, it could be said that this looking at the mathematics semantics, rather than the syntax, is what characterises a computer algebra system.

One fundamental question is where in the computation process the work is done. Some computer algebra systems, notably Macsyma [3] would store the two expressions essentially as parse trees, and the equality operator would do all the work. In many other systems, e.g. Axiom[28] and Reduce [24], the two expressions would be converted on input into data structures representing the underlying mathematics, and it is these data structures that would be compared.

2.1 Formal Treatment

Let us make this more formal. Let \( O \) be a set of mathematical objects (\( \mathbb{Z}[x] \) in the example above), and \( R \) be a set of data structures (parse trees for Macsyma, “standard quotients” for Reduce, etc.).

**Definition 1** A correspondence \( f \) between \( O \) and \( R \) is a representation of \( O \) by \( R \) if:

1. every element of \( O \) corresponds to at least one element of \( R \);
2. every element of \( R \) corresponds to one, and only one, element of \( O \).

In other words, \( f \) has to be a surjective function from \( R \) to \( O \).

Note that this means that \( R \) is the set of valid objects, i.e. those corresponding to elements of \( O \).

**Definition 2** If, furthermore, \( f \) is a bijection, we say that the representation is canonical.

**Proposition 1** If the representation is canonical, then mathematical equality (in \( O \)) is the same as data structure equality (in \( R \)).

In the case where \( O \) is an Abelian group (very common in the polynomial-based systems), there is a weaker concept.

**Definition 3** If \( f \) is a representation such that \( 0 \in O \) has only one representation in \( R \), we say that the representation is normal.

One could ask “why 0?” The reason is that, once one moves on to rings, 0 is the forbidden second input to division. This is, in fact, the reason why equality testing is needed for fast determinant computation (and many other things). So, in fact, a normal representation would suffice in this case.

---

1. Though the syntax might vary. Several systems will in fact regard this as an equation, and some further operation may be required to convert it into a Boolean value.
Proposition 2 If the representation $f$ of $O$ is normal, then $a = b$ (in $O$) if, and only if, $f(a - b) = f(0)$ (in $R$).

Brown [6] proposed the following trick to convert a normal representation $f$ of $O$ by $R$ into a canonical representation $g$ of $O$ by $R' \subseteq R$. In this trick, $R'$ is a dynamically growing set.

(1) Initially, $R' = \emptyset$, and $g = \emptyset$.
(2) As a new object $o$, (with representation $f(o) = r \in R$) is computed, we see if any element $s$ of $R'$ has the property that $f(o - g^{-1}(s)) = f(0)$.
(3) If so, we declare that $g(o) = s$.
(4) Otherwise, we do $R' := \{r\} \cup R'$ and augment $g$ by the corresponding pair $(o, r)$.

Unfortunately, this is not so much a representation of the set $O$ as of the ordered sequence of elements of $O$ as they appear in a particular computation. A variant of this technique, using hashing to reduce the cost drastically, is used in (at least early versions of) Maple [7, section 4].

2.2 Is Equality Determinable?

Canonical representations exist for integers, rational numbers, polynomials, rational functions, and many other data types [14], and these are commonly employed in polynomial-oriented computer algebra systems.

For algebraic numbers, the author knows of no non-order-dependent representation. *Ex post facto*, when one knew all the algebraic extensions required, i.e. the ultimate extension field, one could search for the polynomial of minimal height that generated that field, and represent everything in terms of a root of that, but in practice the search for such a polynomial would be prohibitive [31, Section 3.8]. Most systems make no attempt to produce a canonical, or even normal, representation, and in Maple, for example, it is necessary to invoke simplify explicitly in order to have even a normal representation. However, normal, and indeed order-dependent canonical, representations can be achieved by means of a "growing algebraic tower" strategy [36]. The simplest example of order-dependence is the introduction of $\sqrt{2}$ and $\sqrt{3}$. Depending on the order of introduction, these would be represented as $\sqrt{2}$ and $2\sqrt{2}$, or $\frac{1}{4}\sqrt{2}$ and $\frac{1}{2}\sqrt{3}$.

Elsewhere, we may not be nearly so lucky. Even allowing numbers such as $\sqrt{2}/\sqrt{3}$ means that there is no known test for equality at all, though in some cases careful numerical evaluation can demonstrate inequality. Elementary functions are another problem area; it may not be obvious to the reader (and certainly was not obvious to the author) that

$$\frac{2}{i} \ln \left( \sqrt{\frac{1+z}{2}} + i \sqrt{\frac{1-z}{2}} \right) = -i \ln \left( z + i \sqrt{1-z^2} \right),$$

which proves the equivalence of two different definitions of arccos [8]. Equally, with the definitions of [8], which here mimic the standard ones in [1], it is not the case that

$$\text{arcsin } z = \arctan \frac{z}{\sqrt{1-z^2}},$$

but rather that

$$\text{arcsin } z = \arctan \left( \frac{z}{\sqrt{1-z^2}} \right).$$

Is there an algorithm to verify this last equality?
2.3 Group-theoretic Systems

Much of what has been said above applies largely to the polynomial-based systems (Macsyma, Reduce, Maple, Axiom). When it comes to group theory, we are in much worse shape. A traditional way of defining an abstract group is as a \textit{finitely presented} group, i.e. a (finite) set of generators and a (finite) set of relations between them, such as

\[ \langle a, b | a^2 = 1, b^3 = 1, (ab)^5 = 1 \rangle \quad (4) \]

(a description of the alternating group \(A_5\)).

Unfortunately, the insolubility of the Word Problem [37] for these groups means that there is no general algorithm for determining equality of two elements (written as products of the generators and their inverses), or, equivalently, deciding whether such a product is in fact the group identity. Nevertheless, GAP [38, section 7.1] does provide such a comparison, since if the algorithm terminates at all, it has a definite answer. However, there is scope for confusion here, as exemplified in the following GAP session.

```
gap> F2:=FreeGroup("a", "b");
Group( a, b )
gap> A5:=F2/[F2.1^2,F2.2^3,(F2.1+F2.2)^5];
Group( a, b )
gap> a1:=A5.1;
a
gap> a3:=A5.1^3;
a^3
gap> a1=a3;
false
```

The problem here is that the comparison is essentially taking place in \(F2\) rather than in \(A5\). In GAP 4, a better typing model means that the comparison is carried out, correctly, in \(A5\).

2.4 Fuzzy equality

There are two main families of algebraic structures where a binary true/false is not totally appropriate.

\textbf{Lazy Infinite Structures} This covers a variety of applications where an object is represented as some form of generator, which can generate more accurate approximations as required, either series approximations with a higher \(O(t^n)\) remainder term, or numerical approximations with a smaller guaranteed error bound. The two main examples of this are “lazy power series” [34, 23] and various forms of “lazy generators” for elements of \(\mathbb{R}\) [21, 40, 18, 2, 32], though lazy \(p\)-adics are also possible. The problem in both cases is fundamentally the same: unless one has some extrinsic information, one can develop the series/numbers to as many terms as one wants, and the fact that all the corresponding terms are equal\(^2\) does not prove that the two underlying objects are mathematically equal. Of course, if the objects are not equal, we will eventually prove this.

An example where there is additional information that can help is given by [25, 26] in the case of algebraic numbers represented by lazy generators of their numerical

\(^2\) This works for lazy formal series. For numbers, one should mean “equal within the tolerance allowed at this step”.
value: here the minimal polynomials can be used to deduce bounds such that, if they are equal to within this bound, then they are definitely equal.

In the absence of such information, it is usual to set a limit such that “equal to this length” implies “equal”, but this is clearly mathematically unsound.

**Monte Carlo Methods** There are various cases in computational arithmetic and algebra where one wishes to use a Monte Carlo method; generally defined as an algorithm which, for any fixed $\epsilon$, has a running time polynomial in the size of the input, but may give a wrong answer, with probability less than $\epsilon$. The classic case is Rabin’s [35] primality testing algorithm, where the running time is $O\left(\frac{n^{1.5}}{\log n}\right)$ for numbers of $n$ digits, though in practice\(^3\) the error is much less than that. It is worth noting that the error is one-sided: if the algorithm says “prime” the number might actually be composite, whereas if the algorithm says “composite”, the number definitely is composite.

There are several data structures where we may want to use Monte Carlo algorithms for testing equality, such as polynomials represented as straight-line programs [19] or black-box programs [29]. In both cases what one has is not an explicit formula for the polynomial, but rather an object which, given values for the indeterminates, returns the value of the polynomial at those values. Here deterministic algorithms for equality tend to have exponential running time, and one has to rely, essentially, on an approach which can crudely be described as “if they evaluate to the same thing at enough points, then they are equal.” Again, one can bound the probability of an error, and the error is one-sided, in that there is no possibility of error in an answer of “unequal”. A similar algorithm is used in Maple’s `testeq` [20].

The difficulty is that, *a priori*, one has no idea how often equality testing will be called during the running of an algorithm (and the sub-algorithms it calls). If I want a result which has a 99% chance of being accurate, do I set $\epsilon = 10^{-3}, 10^{-6}$ or what? Some progress towards a solution is described in [33], but it does require a fundamental re-think of how equality is managed, and leads to a choice of strategy: should every object carry with it an indicator that “my Monte Carlo probability is currently $n\%$ (analogous to *a posteriori* error control in numerical analysis), or should each procedure take a parameter which says what the minimum Monte Carlo probability on the result must be, and the procedure has to manage its error analysis accordingly (analogous to *a priori* control)?

### 3 What is “Data Structure Equality”

So far, we have been assuming that this is a well-defined concept. Unfortunately, this is not so, and depends on languages and even on dialects. C, for example, has no concept of data structure equality: two structures are the same if, and only if, they are identical.

#### 3.1 Common Lisp

This [39] defines four equality predicates, listed from most specific to most general.

- `eq` “$x$ and $y$ are the same identical object” (generally implemented as C’s `==`). Depending on the implementation, different copies of the same number may or may not be `eq`.

\(^3\) [9] shows that, for six trials on 256-bit numbers, the error probability is less than $2^{-51}$, rather than the $4^{-6} = 2^{-12}$ that one would expect.
“As eq, except that if \( x \) and \( y \) are characters or numbers (of the same type) their values are compared.

**equal** This compares numbers and characters as for **eq**, symbols as for **eq**, strings and bit-vectors by element-element comparison, other arrays as for **eq**, and it recursively descends **cons** cells. Hence, it can loop on circular structures (which is unfortunate, since these are the usual way of implementing the lazy objects described in section 2.4).

**equalp** This compares characters using **char-equal** (thus ignoring case etc.), numbers irrespective of type (so that (**equalp 2.0 2.0**) is true), and recursively descends **cons**s (like **equal**) and also arrays and hash tables (unlike **equal**). Again, it can loop on circular structures.

### 3.2 IBM’s LISP/VM

This [27] was the Lisp dialect on which Axiom was first developed, and, to the author’s knowledge, the only LISP dialect to tackle the problems of circular structures seriously. It defines three equality predicates, listed from most specific to most general.

**eq** “The objects occupy the same storage [at least conceptually] and a change to one is a change to the other”. This is essentially the same as Common Lisp’s **eq**.

**unequal** The two objects have the same shape (as directed graphs) and the same leaf values. This means that performing the same change to both of them will still leave **unequal** values. It handles circular structures, and takes a time linear in the size of the object. Essentially\(^4\), two objects are **unequal** if no sequences of modifications or accesses can distinguish them. This seems to be a programmatic version of Leibniz’ definition of equality: “two things are equal if we cannot tell them apart”.

**equal** The two objects have the same leaf values, even if the shapes are different. This does handle circular structures, but the (rare) worst-case running time is cubic. Two objects are **equal** if no sequences of accesses (but not modifications) can distinguish them.

Both **equal** and **unequal** descend all kinds of descendable structures — **cons** cells, vectors, hash tables etc. LISP/VM’s **equal** is probably the closest to an abstract definition of “data structure equality”.

### 3.3 The importance of eq

A fact, not widely appreciated, is that **eq** is important in interval arithmetic. In general, intervals do not form even an Abelian group, since

\[
\forall a \neq [a, b] \not\exists [c, d] : [a, b] + [c, d] = [0, 0].
\]  

(5)

If one creates a single interval-valued object, as in \( a := b := [-1, 1] \), then \( a - b = [0, 0] \), since \( a \) and \( b \) must represent the same unknown number in \([-1, 1]\). Conversely, if we have two different interval-valued objects, as in \( a := [-1, 1] \) and \( b := [-1, 1] \), then \( a - b = [-2, 2] \), since now \( a \) and \( b \) might represent different numbers in \([-1, 1]\). Hence, despite (5), it is possible to say that \( a - a = [0, 0] \) for any interval \( a \), but not via the intervention of some putative \(-a\). It is also pointed out in [11] that evaluating polynomials at intervals requires a careful consideration of **eq-ness**.

\(^4\) There are bizarre counter-examples when the structures overlap with each other, unfortunately.
Similar concepts apply for other data types for which equality in general is difficult, such as infinite (lazy) power series, infinite-precision real numbers, black box objects, and so on.

4 Equality in higher languages

ML defines equality to be data structure equality (no formal comment is made about circular structures). This fact has hampered some efforts to build computer algebra systems in ML.

There is no requirement in Axiom [28] or Magma [4, 5] for data structure equality to be the same as mathematical equality. The designer of a data type can define any definition of equality; for example, in defining finitely presented groups as in equation (4), the equality algorithm would have to take account of the relations (via the Todd-Coxeter algorithm or an equivalent). In Axiom, types for which data structure equality is known to be same as mathematical equality are said to be canonical (in line with the definitions in section 2.1). This causes substantial problems [30] when interfacing with proof-checkers where the two are assumed to be the same, and a full implementation of, say, Axiom's logic in Larch [22], would need to distinguish between the two kinds of equality.

Whether or not an Axiom data type is canonical can be quite complex. For an example, we consider the constructor Fraction, which builds the field of fractions of an integral domain \( R \). For Fraction(\( R \)) to be canonical, the following conditions have to be satisfied [12]:

- \( R \) itself must be canonical;
- \( R \) must be a GcdDomain;
- \( R \) must possess a function (called unitNormal) which chooses a canonical representative of any non-zero \( r \in R \) and its associates.

This last is necessary to implement the appropriate generalisation of the usual rule that the denominator of a fraction is always positive.

Oddly enough, it seems hard to find any description of equality in the on-line documentation for Maple. This may be because the semantics are not totally clear — the author certainly found it hard to work out what they might be.

5 Conclusion

We hope that we have demonstrated that there is more to equality than meets the eye. Data structure equality (whatever that means) may or may not be the same as mathematical equality. In particular, it seems to us to be important to distinguish carefully (unlike Larch and ML) between the two in principle, even though they may often be the same in practice. When interfacing two systems, such as Axiom and Larch [30], it can be important to understand the subtle differences that may exist.

We therefore have four concrete recommendations.

(1) Anyone using multiple mathematical systems should take great care that differences in the meaning(s) of equality are allowed for.

(2) Conversely, the designers of individual systems should make sure that the meaning(s) of equality in their systems are clearly documented.
(3) In particular, more clarity is needed over the phrase “data structure equality”, and its interaction with any form of storage model.

(4) As a more minor, but practical, point of implementation, can we plead for system and language designers to understand circular structures — see section 2.4?

References


Extensions to Proof Planning for Generating Implied Constraints

Alan M. Frisch, Ian Miguel and Toby Walsh

Artificial Intelligence Group
Department of Computer Science
University of York
York, England
frisch.iam.tw@cs.york.ac.uk

Abstract Extending a constraint satisfaction problem with additional constraints that are implied by those already in the problem formulation can often result in a marked reduction in the amount of search needed to solve the problem. Consequently, experts often add such constraints to their problem formulation. This paper shows how we are using and extending proof planning techniques to derive such constraints automatically, particularly in a domain of algebraic constraints. This inference problem introduces a number of challenging problems like deciding a termination condition and evaluating constraint utility. We have implemented a number of methods for reasoning about algebraic constraints. For example, the eliminate method performs Gaussian-like elimination of variables and terms. We are also re-using proof methods from the PRESS equation solving system such as (variable) isolation.

1 Introduction

Users of computer algebra systems typically have well-defined goals. For example, they might wish to find the solutions to some algebraic equations, or factorize a polynomial. We are interested in an inference problem about algebraic constraints which is less well-defined. We wish to infer some implied constraints (logical consequences of the initial problem representation) that will help a constraint solver. Adding implied constraints can lead to a marked reduction in search [SSW99], but this process is usually performed by hand.

It is difficult to know how many implied constraints to infer, and which will be useful to a particular constraint solver. To tackle this problem, we are using Bundy’s proof planning framework [Bun91]. This is one of the most promising inference techniques for dealing with combinatorially explosive search through a space of inferences. Proof plans are built by methods which come with strong preconditions to limit their applicability. Some of the proof methods we are developing are extensions of those used by the PRESS equation solving system [BW81]. Others of the proof methods are novel, and perform tasks like eliminating variables and linearizing constraints.

The paper is structured as follows. In Sections 2 and 3, we introduce proof planning, and describe how it has been extended to deal with generating implied constraints. In Section 4, we describe the proof methods currently implemented that infer implied constraints. In Section 5, we illustrate their behaviour on two examples taken from the
literature. In Section 6, we describe related work. We end in Section 7 with future work and conclusions.

2 Proof Planning

Proof planning is a technique used for guiding the search for a proof in automated theorem proving [Bun91]. Common patterns in proofs are identified and encapsulated in methods which are made available to a planner. Methods have strong preconditions which limit their applicability and prevent combinatorially explosive search. Proof planning has often been associated with “rippling” [BSvH’93], a powerful heuristic for guiding search in inductive proof. However, proof planning can easily be adapted to other mathematical tasks like finding closed form sums to series [WNB92] or, as here, generating logical consequences which may make useful implied constraints.

A proof planner like CLAM [BvHHS90] takes a goal to prove, and selects from a database of methods a method which matches this goal. Pre-conditions of the method, each of which is a sequence of statements in a meta-logic, are checked. If the pre-conditions hold then the proof planner executes the post-conditions (which are also sequences of statements in the meta-logic). This constructs the output goal or goals. Associated with each method is a tactic which applies individual inference rules to construct the actual proof. A typical method is the induction method, whose input is an universally quantified goal, and whose preconditions then select a suitable induction variable, and induction scheme. The induction method outputs an appropriate base case and step case. Proof planning terminates when all the goals have been satisfied.

Proof planning offers several potential advantages over other theorem proving techniques for the task of generating implied constraints automatically. In particular, method-based operation enables a planning level beyond what can be achieved via the procedural application of tactics alone. First, methods can be given very strong preconditions to limit the generation of logical consequences to those that are likely to make useful implied constraints. Second, methods can act at a very high level. For example, they can perform complex rewriting, simplifications, and transformations. Such steps might require very long and complex proofs to justify at the level of individual inference rules. And third, the search control in proof planning is cleanly separated from the inference steps. We can therefore easily try out a variety of different search strategies like best-first search or limited discrepancy search.

3 Extensions to Proof Planning

Whilst proof planning has a number of features that make it well suited to the task of generating implied constraints, we have extended it along a number of dimensions to deal with the following issues:

Non-monotonicity: Our proof methods transform one set of constraints into another. In some cases, they might add a new constraint. In others, they might replace one constraint by a tighter one, or eliminate a redundant constraint. The set of constraints may therefore increase or decrease. To deal with this, we replace the “output” slot in a method by the “add” and “delete” lists used in classical planning.

Pattern matching: Existing proof planners like CLAM [BvHHS90] use first-order unification to match a proof method’s input against the current proof goal or subgoal. We use a richer pattern matching language specialized to the task of
reasoning about sets of constraints. For example, the input to a proof method is a set of constraints, and this is matched against any subset of the initial or inferred constraints.

Looping: Unless a proof method deletes one (or more) of the input constraints, the preconditions of the method will typically continue to hold. Proof methods may therefore repeatedly fire, generating identical implied constraints. We therefore added a history mechanism to the proof planner to prevent such repeated method application.

Constraint utility: Deciding which implied constraints will be useful to a constraint solver is also very difficult. The proof planner uses measures like constraint arity and tightness to eliminate implied constraints which are likely to be useless. However, it remains difficult for the proof planner to decide which of the remaining constraints to keep. We are therefore inventing some heuristics to decide which of the inferred constraints to give to the constraint solver.

Termination: Previously, proof planning had a clear termination condition. We reduced a goal to subgoals, and when all these had been proven, we finished. The termination condition is much less clear when using proof planning to infer implied constraints. There are many logical consequences (including the solution to the problem) which could be inferred. At some point, we must decide to stop inferring new constraints and start searching for an answer. At present, our methods have strong enough preconditions that we can run them till exhaustion. However, we may in the future have to add an executive along the lines of Ireland's proof critics [Ire92], which terminates proof planning when future rewards look poor.

Explanation: Methods, as defined in [Bun91], do not explain what they do. In order for the user to see how an implied constraint was generated, we adapted the tactic mechanism already used within proof planning. Tactics now write out text explaining the application of the methods.

4 Methods

The set of methods described here is not complete—we plan to add more as the system is developed. However, it is sufficient to illustrate the operation of the proof planner.

The following example is taken from the implied constraint section of the Oz finite domain constraint programming tutorial [SS1]. We wish to find 9 distinct non-zero digits, \( A \) to \( I \), which satisfy the constraint:

\[
\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1
\]

(1)

Note that \( BC \) is shorthand for \( 10 \ast B + C \), \( EF \) for \( 10 \ast E + F \) and \( HI \) for \( 10 \ast H + I \).

4.1 Symmetry method

The first method to fire is often the symmetry method. The preconditions to the symmetry method identify variables or terms which are indistinguishable. In the former case, if swapping the variable \( x \) for the variable \( y \) (and vice versa) gives the same set of constraints, then \( x \) and \( y \) are indistinguishable. In the latter case, all pairs of corresponding variables within two terms are swapped and the same check is made.

\[\text{http://www.mozart-oz.org/documentation/fdt/node31.html#section.propagators.fractions}\]
To break this symmetry, the symmetry method adds an ordering constraint that puts an order on the indistinguishable variables or terms. In the above case, it adds the constraints that \( x \leq y \). This is not an implied constraint since it does not follow from the initial model. However, symmetry breaking constraints are very useful both for reducing search, and, as we show in the next section, for generating other implied constraints using the eliminate method. Note that the symmetry method preserves satisfiability: all other solutions can be generated by permutation.

Identifying symmetries, especially between non-atomic terms, is potentially expensive. We are therefore developing heuristics to identify terms for comparison that are likely to be symmetrical. These heuristics are based primarily on structural equivalence. Two terms are said to be structurally equivalent if they are identical when explicit variable names in each are replaced with a common indistinguishable ‘marker’. For example,

\[
\frac{A}{BC} \leq \frac{D}{EF}
\]

become:

\[
\frac{A}{BC} \leq \frac{D}{EF} \leq \frac{G}{HI}
\]

and are therefore structurally equivalent. Each pair of variables, \( A \) and \( D \), \( B \) and \( E \), and \( C \) and \( F \) are swapped throughout the problem definition and the indistinguishability test is made.

The symmetry method applied to the initial problem definition of the fractions puzzle identifies the fact that the three fractions are indistinguishable and breaks the symmetry by adding the constraints:

\[
\frac{A}{BC} \leq \frac{D}{EF} \leq \frac{G}{HI} \quad (2)
\]

We are currently investigating methods for identifying and breaking other forms of symmetry like rotations and reflections.

### 4.2 Eliminate method

The next method to fire is often the eliminate method. This uses symmetry breaking constraints, as well as other equations and inequalities, to perform Gaussian-like variable elimination. The preconditions to the eliminate method identify variables or terms which can be eliminated from a non-linear constraint. This gives an implied constraint of smaller arity than the original non-linear constraint. As constraint solvers typically delay non-linear constraints until their variables are ground, the eliminate method generates an implied constraint which may be used by a constraint solver at an earlier point in its search.

For example, in the fractions puzzle, eliminate can be used to simplify equation (1) using equation (2) by eliminating first \( \frac{G}{HI} \) and then \( \frac{D}{EF} \) in favour of \( \frac{A}{BC} \) as follows:

\[
\frac{A}{BC} + 2 \frac{D}{EF} \leq 1 \quad (3)
\]

\[
3 \frac{A}{BC} \leq 1 \quad (4)
\]

Similarly, by eliminating first \( \frac{A}{BC} \) and then \( \frac{D}{EF} \) in favour of \( \frac{G}{HI} \) produces:

\[
2 \frac{D}{EF} + \frac{G}{HI} \geq 1 \quad (5)
\]

\[
3 \frac{G}{HI} \geq 1 \quad (6)
\]
Equations (4) and (6) are both ternary, as opposed to the original arity 9 constraint and are therefore much more likely to be useful for pruning earlier in the search.

The \texttt{eliminate} method uses both equations and inequalities to rewrite constraints. When rewriting with inequalities, it computes the polarity of the terms being rewritten based on the monotonicity properties of the algebraic operators [Sch99]. For instance, addition is monotonic in both of its arguments as replacing either argument with a larger number increases the sum. On the other hand, division is monotonic in the numerator argument but anti-monotonic in the denominator argument; replacing the numerator with a larger number increases the fraction, whilst replacing the denominator with a larger number decreases the fraction.

### 4.3 Linearise method

As mentioned before, constraint solvers typically delay non-linear constraints until variable instantiations make the constraint linear. A \texttt{linearise} method therefore attempts to infer linear constraints from non-linear constraints as these will be of more use to a constraint solver. The preconditions to the \texttt{linearise} method identify non-linear constraints which can be converted to linear constraints by cross-multiplying terms. We are currently investigating other ways to linearise terms.

The \texttt{linearise} method can be applied effectively to equations (4) and (6) to produce:

\begin{align*}
3A & \leq 10B + C \quad (7) \\
3G & \geq 10H + I \quad (8)
\end{align*}

Bounds consistency applied to (7) and (8) can be start pruning values even before search starts.

### 4.4 All-different method

Many problems have a constraint that certain variables must take distinct values. For example, the times of classes assigned to a particular teacher must all be different from each other. A specialized \texttt{all-different} method reasons about constraints containing variables which take distinct values. The preconditions to the \texttt{all-different} method identify variables in a summation or product constraint which take distinct values. The method then computes upper and lower bounds based upon the variables taking distinct values. We are again looking at extending the method to other types of constraints.

The \texttt{all-different} method produces the following when applied to equations (7) and (8) and the all-different constraint:

\begin{align*}
12 & \leq 10B + C \leq 98 \quad (9) \\
12 & \leq 10H + I \leq 98 \quad (10)
\end{align*}

This is because the lower bound of each variable is 1, but since they are all-different, the least combination of values is 1 and 2 in both cases. Similar reasoning gives the upper bound. An application of \texttt{eliminate} to (8) and (10) gives a tight lower bound for $G$:

\begin{equation}
G \geq 4 \quad (11)
\end{equation}

Note that, on this occasion, bounds consistency establishes the same lower bound on $G$ using equation (8) because of integer division on $3G \geq 11$. In general, however, when more all-different variables are involved, this method can be expected to do significantly better than naive bounds consistency.
4.5 Introduce method

The introduce method is complimentary to the eliminate method as it introduces a new variable into a constraint. The preconditions to the introduce method identify a non-atomic subterm common to two (or more) constraints. It then introduces a new variable for this common subterm. As the introduced variable is common to two (or more) constraints, this tightens the constraint graph. This can lead to increased propagation within a constraint solver. For example, in the Golomb rulers problems, Smith et al [SSW00] introduce auxiliary variables for the inter-tick distances as these were common to a large number of constraints. We are also considering methods to introduce new variables equal to the sum of certain other variables. This is a common trick for reasoning about slack or wastage. For example, introducing a new variable equal to sum of some of the existing decision variables on a circular Golomb ruler problem produces a significant reduction in search [SSW00].

5 Other methods

In many examples, we have found that we may have to do some algebraic manipulations in order to eliminate some subterm. We currently use the isolate method from the PRESS equation solving system [BW81] to isolate a variable on one side of an equation or inequality. We also see uses for other methods from PRESS like collect and attract.

In addition, some methods may produce output which could be usefully simplified before being added to the current set of constraints. A simplify method is therefore used to perform certain simplifications, again using methods such as isolate and, in the future, collect and attract. A normalise method is also applied to the output of other methods in order to maintain a normal form on the set of equations the proof planner is working on. This helps to avoid the same constraint being added twice (in slightly re-arranged form).

6 Results

We illustrate the behaviour of these methods on two examples from the literature. In addition to listing the implied constraints generated, we show the reductions in runtimes using the SICSTUS finite domain constraint solver.

6.1 Fractions puzzle

The proof planner takes as input a set of constraints specifying the problem. For example, the input for the fractions puzzle is given below:
\[
\text{prob('Fractions Puzzle',}
\begin{align*}
\text{domain(var('A'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('B'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('C'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('D'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('E'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('F'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('G'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('H'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{domain(var('I'), [1, 2, 3, 4, 5, 6, 7, 8, 9]),} \\
\text{all_different([var('A'), var('B'), var('C'), var('D'), var('E'),} \\
\text{var('F'), var('G'), var('H'), var('I')]),} \\
\text{eq(var('A'))/(10*var('B')+var('C')) +} \\
\text{var('D')/(10*var('E')+var('F')) +} \\
\text{var('G')/(10*var('H')+var('I'))} \\
\end{align*}
\]

In the future, we intend to accept input in a high level constraint modelling language like OPL or ESRA [FH01].

The proof planner outputs a new problem formulation—the original constraints with the identified additions and deletions made. Furthermore, associated with each method is a tactic which explains how the implied constraint generated by the method is inferred. In the future, we intend to output \LaTeX{} and HTML as well as plain ASCII text. Part of the proof planner's output on the fractions puzzle is given below:

Using \(\frac{\text{var}(A)}{10\cdot\text{var}(B)+\text{var}(C)} \leq \frac{\text{var}(D)}{10\cdot\text{var}(E)+\text{var}(F)}\), we eliminate \(\text{var}(A)/(10\cdot\text{var}(B)+\text{var}(C))\) in favour of \(\text{var}(D)/(10\cdot\text{var}(E)+\text{var}(F))\) in

\[
\text{var}(A)/(10\cdot\text{var}(B)+\text{var}(C)) + \text{var}(D)/(10\cdot\text{var}(E)+\text{var}(F)) + \text{var}(G)/(10\cdot\text{var}(H)+\text{var}(I)) = 1.
\]

This gives: \(1 = 2\cdot(\text{var}(D)/(10\cdot\text{var}(E)+\text{var}(F))) + \text{var}(G)/(10\cdot\text{var}(H)+\text{var}(I))\).

Using \(\text{var}(D)/(10\cdot\text{var}(E)+\text{var}(F)) \leq \text{var}(G)/(10\cdot\text{var}(H)+\text{var}(I))\),
we eliminate \(\text{var}(D)/(10\cdot\text{var}(E)+\text{var}(F))\) in favour of \(\text{var}(G)/(10\cdot\text{var}(H)+\text{var}(I))\) in

\[
1 = 2\cdot(\text{var}(D)/(10\cdot\text{var}(E)+\text{var}(F))) + \text{var}(G)/(10\cdot\text{var}(H)+\text{var}(I)).
\]

This gives: \(1 = 3\cdot(\text{var}(G)/(10\cdot\text{var}(H)+\text{var}(I)))\).

Linearising \(1 \leq 3\cdot(\text{var}(G)/(10\cdot\text{var}(H)+\text{var}(I)))\)
\text{Gives: } 10\cdot\text{var}(H)+\text{var}(I) \leq 3\cdot\text{var}(G)

Since we know that the variables in \(10\cdot\text{var}(H)+\text{var}(I)\)
are all-different, the lower bound of this summation is 12,
and the upper bound 98.

Using \(12 \leq 3\cdot\text{var}(G)\),
we eliminate \(10\cdot\text{var}(H)+\text{var}(I)\) in favour of 12 in

\[
10\cdot\text{var}(H)+\text{var}(I) \leq 3\cdot\text{var}(G).
\]

This gives: \(4 \leq \text{var}(G)\).
The proof planner generates 22 implied constraints in this manner. Even with such
verbose output, this takes less than 1 second. In the context of the simple example
problems, this is significant, but in general we can expect that the time taken by the
proof planner will be negligible compared to the time required to solve the problem.

We are currently developing heuristics to prune the output constraint set. One of
the simplest heuristics is the constraint arity. If we delete all implied constraints of
arity 4 or greater, we get the following five constraints (including 2 symmetry-breaking
constraints) to add to the problem definition:

\[
\begin{align*}
\frac{A}{BC} & \leq \frac{D}{EF} \\
\frac{D}{EF} & \leq \frac{G}{HI} \\
G & \geq 4 \\
3G & \geq 10H + I \\
3A & \leq 10B + C
\end{align*}
\]

Table 1 presents the results obtained when solving the fractions problem with the
SICSTUS finite domain constraint library. The first column gives the results using
the basic model only, the second shows the result of adding the small set of implied
constraints described above, and the third shows the results of adding all the implied
constraints generated by the proof planner. In both cases, the implied constraints
provide a significant reduction in backtracking during search.

The 22 implied constraint model takes longer to solve because of the overhead
involved in maintaining consistency in all the extra constraints. However, it also sig-
ificantly reduces the size of the search tree. This suggests that there is a mid-point
between the 2 which out-performs them both. In [SS] Schulte and Smolka report that
adding the symmetry breaking and implied constraints to their model of the fractions
puzzle reduces the size of Oz’s search tree by one order of magnitude.

<table>
<thead>
<tr>
<th></th>
<th>Basic Model</th>
<th>Basic Model+ (5) Implied Constraints</th>
<th>Basic Model+ (22) Implied Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backtracks (1st solution)</td>
<td>3203</td>
<td>2689</td>
<td>1529</td>
</tr>
<tr>
<td>Time Taken(ms) (1st solution)</td>
<td>1450</td>
<td>1280</td>
<td>2460</td>
</tr>
<tr>
<td>Backtracks (all solutions)</td>
<td>13350</td>
<td>3556</td>
<td>2059</td>
</tr>
<tr>
<td>Time Taken(ms) (all solutions)</td>
<td>5470</td>
<td>1690</td>
<td>3310</td>
</tr>
</tbody>
</table>

Table 1: The Fractions Problem: Results
6.2 Professor Smart's Safe

This example problem is also taken from the Oz finite domain constraint programming tutorial\(^2\). The code of Professor Smart's safe is a sequence of 9 non-zero digits \(x_1, \ldots, x_9\) such that the following constraints are satisfied:

\[
\begin{align*}
  x_4 - x_0 &= x_7 \\
x_1 x_2 x_3 &= x_8 + x_9 \\
x_2 + x_3 + x_6 &< x_8 \\
x_9 &< x_8 \\
x_i &\neq i
\end{align*}
\]

all-different\((x_1, \ldots, x_9)\)

The proof planner produces the following set of additional constraints:

\[
\begin{align*}
  2x_9 &< x_1 x_2 x_3 < 2x_8 \\
  6 &\leq x_1 x_2 x_3 \leq 504 \\
  6 &\leq x_2 + x_3 + x_6 \leq 24 \\
  3 &\leq x_6 + x_7 \leq 17 \\
  3 &\leq x_8 + x_9 \leq 17 \\
  6 &< x_8 \\
  3 &< x_4
\end{align*}
\]

Notice that the implied constraints reduce the domains of possible values for the variables. For example, the search algorithm need only consider assigning value 7 or 9 to variable \(x_8\) because of the constraints \(6 < x_8\) and \(x_8 \neq 8\).

Table 2 presents the results of comparing the basic model and the basic model with implied constraints. Run-times are negligible and so are not reported. Even though this problem in its basic state is very easy to solve, the addition of the implied constraints still gives some improvement.

<table>
<thead>
<tr>
<th></th>
<th>Basic Model</th>
<th>Basic Model + Implied Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backtracks</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(1st solution)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backtracks</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>(all solutions)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Professor Smart's Safe: Results

\(^2\) http://www.mozart-oz.org/documentation/fdt/node19.html#section.problem.safe
7 Related Work

Proof planning has been implemented in a number of systems. The CLAM proof planner developed in Edinburgh controls search in the Oyster proof checker [BvHHS90]. CLAM has also been linked to the HOL theorem prover [BSB98]. Whilst much of the development of CLAM has been for inductive proof, several other domains have been explored including finding closed form sums to series [WNB92]. Prior to CLAM, the PRESS system used a meta-level representation of proof methods to solve algebraic equations [BW81]. Despite the lack of a planner to put its methods together, PRESS was competitive with computer algebra systems of its era. The Ω system developed in Saarbrücken also implements proof planning, but in this case for a higher order natural deduction style logic [HKK+94]. It also contains some of the extensions we found necessary for generating implied constraints, such as richer pattern matching and the ability to add and delete conclusions and premises.

Recently, the third author has built a simple proof planning shell, CLAM-Lite, on top of the Maple computer algebra system [Wa00]. This system allows us to explore how proof and computation can be mixed together. It can, for example, find a closed form sum to a series, and prove by induction that the answer is correct.

As exemplified by [SSW99], several recent studies show that implied constraints added by hand to a problem representation can lead to significant reductions in search. However, outside a highly focused domain like planning (see, for example, [EMW97]), there has been little research on how to generate such implied constraints automatically. One exception is [Fri99], which generalises resolution to multi-valued clauses in which variables can take more than just the two values True and False, and proves that implied constraints generated by the closure of this operation will eliminate search.

A number of other projects have combined tools for performing inference and algebraic reasoning. For example, the Theorema project [BJK+97] is extending the Mathematica computer algebra system with theorem proving capabilities. The system consists of a collection of special purpose provers. These include a prover for induction over the natural numbers, another for induction over lists, as well as an interface to external theorem provers. The Analytica prover [BCZ96] also adds theorem proving capabilities to the Mathematica computer algebra system. The system is able to prove some complex theorems in analysis about sums and limits, as well as some simple inductive theorems.

8 Future Work and Conclusions

We have described a new application for proof planning, the generation of implied (algebraic) constraints. This required a number of extensions to proof planning like the inclusion in methods of add and delete lists (as in classical planning). We have implemented a number of proof methods for generating implied constraints automatically including: the symmetry method which breaks symmetries between indistinguishable variables and terms, the eliminate method which eliminates variables and terms from non-linear constraints, the introduce method which introduces new variables, the linearize method which linearizes non-linear constraints, and the all-different method which reasons about constraints containing variables which take distinct values.

In future work, we will test these methods on a larger corpus of examples. In addition, we intend to develop a number of specialized methods. For instance, in many scheduling, partitioning and cutting problems, new variables are introduced equal to sums of certain decision variables (for example, to compute the tardiness or wastage)
and implied constraints inferred about these variables. We will implement methods to
perform such variable introduction and inference.

Acknowledgements

This project and the second author are supported by EPSRC Grant GR/N16129\(^3\). The third author is supported by an EPSRC advanced research fellowship. The authors wish to thank Brahim Hnich for discussions about generating implied constraints automatically. Brahim Hnich and Julian Richardson's adaptation of PRESS to manipulate inequalities and its success at generating a number of implied constraints automatically [HRF00] helped inspire the search for the methods presented here.

References


\(^3\) http://www.cs.york.ac.uk/aig/projects/implied/index.html


Some hints for polynomials in the FOC project

S. Bouhme, T. Hardin, and R. Rioboo

Laboratoire d’Informatique de Paris 6 (LIP6), Université Pierre et Marie Curie (Paris 6), 8, rue du Capitaine Scott, 75015 Paris, France

Abstract The Foc project aims at supporting, within a coherent software system, the entire process of mathematical computation, starting with proved theories, ending with certified implementations of algorithms. In this paper, we explain our design requirements for the implementation, using polynomials as a running example. Indeed, proving correctness of implementations depends heavily on the way this design allows mathematical properties to be truly handled at the programming level.

1 Introduction

The Foc project started in 1997 is currently building a development environment for certified computer algebra, that is, a framework for programming algorithms, proving their mathematical properties and the correctness of their implementations. This is a long-term project as its aims may seem rather ambitious. Indeed to ensure the correctness of implementation of mathematical algorithms one needs to formalize the underlying mathematical theories, to formalize the semantics of the different programming constructions and to create tools for proofs. We do not want to embrace all mathematics at a time and we focus first on computations on polynomials, choosing the sub-resultant algorithm as our running example. In this paper, we report on our implementation on polynomials, trying to explain how our objectives of certification have influenced our choices.

Computer Algebra Systems (CAS in short) perform exact (or symbolic) computations on mathematical entities which are represented by terms of a formal language. The correctness (and the mathematical meaning) of the algorithms underlying these computations are ensured by the proofs given by mathematicians. Despite this care, bugs are not so rare [18]: algorithmic errors (hasty simplifications, required assumptions which are not actually discharged, etc.), implementation errors (incorrect typing, bad management of inheritance, bad deallocation, etc.). Indeed, data manipulated by computer algebra programs are quite huge (polynomial coefficients with several thousands of digits), computations may be long (several hours of CPU time is common). Once the proofs of algorithms are written down, there remains a lot of work to choose the appropriate data structures and coding of algorithms. For example, even if explicit manipulation of pointers is recognized to be error-prone (it is banished for strongly critical software), it is often used inside CAS to improve computation times or data representation management.

\footnote{F for formel i.e. symbolic in French, O for Ocaml, C for Coq\cite{6}}
In software engineering, it is now admitted that formalizing specifications and proving required properties directly on these specifications is an efficient way to increase safety. Then a careful coding in a semantically sound (part of a) programming language is required and correctness proofs on code reinforce confidence. Parts of the code may also be extracted automatically from this formalization. To follow the same trail, besides the formalization of algebraic structures, we have to reduce the gap between the mathematical abstract description of an algorithm and its effective implementation.

It is rather easy to describe formally a given mathematical structure (i.e. a set endowed with operations and properties). But deciding on what are the primitive notions and what are the derived ones depends on mathematical habits. This may have a very practical influence on further proofs (see for example the formalization of partiality done in Coq[8]).

Often, this mathematical structure depends on previously defined ones, leading to the need for inheritance mechanisms, which have to be semantically described, up to the question of late binding, allowing the user to replace an inherited code by a new one.

As well-known by people working on formalization of mathematics, even if there is no implementation, the specification of fragments of mathematics requires complex representation choices: how to express dependencies among the underlying sets? are functions first-class citizen? which equality on functions? etc. Dealing with true coding of algorithms adds several specific problems. For example, one wants to have a general notion of polynomials, allowing sharing of properties and of some algorithms but one wants also to firmly distinguish between two different implementations (sparse and dense for example) of polynomials because correctness proofs rely heavily on the data representation.

Our first requirement is the following. The library of algebraic structures has to provide not only the implementation of the classical tools to manipulate algebraic structures, but also their semantics, given by explicit verified statements. To code a given algorithm, the user of Foc may freely use elements of this library, prove properties of this algorithm, define an implementation and prove its correctness. This needs a strong interaction between programming and proving. This was reflected at the very beginning of the Foc project by the choices we were led to. We could either ease the design of the proof part or of the programming part. For example, we could have chosen to code all the algorithms as Coq functions, code being extracted from the proofs. Then, mathematical description of algebraic structures would have been (rather) simple and encoding of algorithms (rather) close to their specification. But efficiency in Computer Algebra remains a bottleneck. Thus, we chose to focus on the programming part, trying to express as much as possible mathematical properties inside the programming language. This led us to some requirements upon this programming language and upon the ways of writing code. At the end, we have chosen the programming language Ocaml as it fits well with our requirements and it is also the development language of the proof language Coq. Some other languages like CASL, Haskell, etc. may fit our requirements as well and it will be interesting to try an implementation with such programming or specification languages in the future, following the same trails.

In this paper, we present our choices for the specification and the implementation of the Foc library, focusing on the programming part but pointing out all the informations which have to be reflected on the proving side. Section 2 presents the encoding of mathematical structures. The example of polynomials is almost completely given in section 3, starting from an abstract view of univariate polynomials, giving then a sparse representation and ending with the recursive representation of multivariate polynomials.
2 Specifying mathematical structures

In this section we discuss the encoding of a mathematical specification of algebraic structures in a programming language. We first recall some basic algebraic definitions needed to describe our running example of polynomials (we apologize to our mathematician readers). Then, we set up our requirements for the encoding and we end with some illustrating code, written in Ocaml. The code is commented so we hope that it remains understandable for the reader, even if not acquainted with this language.

2.1 Algebraic background

A ring[13] $A$ is a set with a binary addition and a binary multiplication. Usually addition will be denoted $+$, and, if ambiguous, by writing $+_A$ when necessary. Multiplication will be denoted by $*$ or by $*_A$ if ambiguity. We thus provide the set $A$ with two binary operations : $+_A : A \times A \rightarrow A$ and $*_A : A \times A \rightarrow A$. These operations have some properties :

- $(A, +)$ is an additive abelian group that is
  - $+$ is associative : $\forall x, y, z \in A, x + (y + z) = (x + y) + z$. To express this property, an equality is needed. Thus the set $A$ should have an equality $= (\equiv_A)$.
  - $+$ has a neutral element $0$ (often disambiguate by $0_A$) in $A : \forall x \in A, x + 0 = 0 + x = x$. These two properties give to $(A, +)$ the structure of a monoid.
  - Every element of $A$ has an opposite in $A : \forall x \in A, \exists y \in A, x + y = y + x = 0$. With this property, $(A, +)$ is a group. Thus a group has all operations and properties of a monoid. This is a form of inheritance.
  - $+$ is commutative : $\forall x, y \in A, x + y = y + x$ this property states that the group is abelian$^2$.
- $(A, *)$ is a multiplicative monoid whose neutral element$^3$ is denoted by $1$ or $1_A$.
- multiplication is left and right distributive with respect to addition, that is $\forall x, y, z \in A, x(y + z) = xy + xz$ for left distributivity and $\forall x, y, z \in A, (x + y)z = xz + yz$ for right distributivity.
- the ring is said to be commutative if its multiplication is commutative.

Now, let $(M, +_M)$ be an additive abelian group and let $(A, +_A, *_A)$ be a ring, we say that $M$ is a left $A$-module if we have an external multiplication $*_M$ between elements of $A$ and elements of $M$. This multiplication should be compatible with operations on $A$, meaning that

- $\forall a, b \in A, \forall m \in M, (a +_A b) *_M m = (a *_M m) +_M (b *_M m)$
- $\forall a, b \in A, \forall m \in M, (a *_A b) *_M m = a *_M (b *_M m)$
- $\forall a \in A, \forall m, n \in M, a *_M (m +_M n) = (a *_M m) +_M (a *_M n)$
- $\forall m \in M, 0_A *_M m = 0_M$
- $\forall m \in M, 1_A *_M m = m$

Similar operations and properties hold for a right $A$-module with external multiplication denoted by $*_M$. An $A$-module is a left and right $A$-module. We thus see that under the common name $+$ we have two distinct operations $+_A$ and $+_M$ and that under the common name $*$ we have three distinct operations $*_A$, $*_M$, and $*_M$. Some choices have to be done for the management of this overloading in mathematical notation.

$^2$ we reserve the word commutative for a multiplicative monoid
$^3$ Some authors do not require the existence of a neutral element.
We say that $E$ is an $A$-algebra if it is an $A$-module which has a $A$-bilinear mapping denoted by $*$. If $A$ and $B$ are two rings and if $f$ is a ring morphism from $A$ to $B$, $B$ may be viewed as an $A$-algebra if the sub-ring $f(A)$ of $B$ commutes with $B$. We define the $A$-module external multiplications by $a * b = f(a) * b$ and $b * a = b * f(a)$. As usual in commutative algebra we restrict our view to this particular type of algebras. We will often consider rings to be commutative by default and we require the ring morphism $f$ to be injective allowing to view $A$ as a sub-ring of $B$.

2.2 Requirements for encoding mathematics

A CAS manipulates entities such as integers, polynomials, which are elements of some set. Thus we need a representation of these entities. What kind of representation? In mathematics sets are often considered as containers for elements. A set may have algebraic properties, such as being a ring. Denotations of the elements are needed to formulate these properties. But, no concrete information on these elements is required, apart from the existence of the membership relation and an equality.

To decrease distance between mathematics and code, tools to encode abstract views of representations are required. There are several ways to do that.

A first one is given by a (naive) object paradigm, described in most of textbooks on classes. Then object-oriented features like inheritance allow to say for instance that a ring is built upon an additive abelian group.

Entities are considered as objects of some class $C_A$ encoding the set $A$. A distinguished entity, as the unit of a group, can hardly be encoded as an object because objects are created only at run-time. So it is considered as a nullary operation, which is mathematically correct, but may demand the establishment of a conversion morphism when proving algorithms.

Operations are encoded by methods. For instance to add two elements $a$ and $b$ of a group $G$, one would write $a +_G b$ in mathematics whereas one should send the method $+_G$ to the object $a$ of the class $G$ with the argument $b$. Then arity is lost, we encounter the binary method problem, well-identified in programming languages. This is hardly acceptable on the proof side: code has to remain close to specifications and the arity of functions must be kept, at least for practical reasons.

We can adopt a more declarative view of mathematical structures. They are a bunch of operations acting over some sets and having some properties. The representation of entities is simply one part of the definitions. This is typically the view proposed by languages of the abstract data types framework, where types can be parameters or more defined expressions. Following this view, we may encode a set by a type, here after called the carrier of the set. Operations on a set can be encoded as functions on the carrier. For instance if a set $A$ has a binary additive law this can be encoded by a function named $+$ having type $\tau_A \rightarrow \tau_A \rightarrow \tau_A$ if the carrier for $A$ is $\tau_A$. A constant of $A$ can be some named data of type $\tau_A$, for instance a neutral element for a binary additive law can be a constant $0$ of type $\tau_A$.

The carrier and the operations defining a given structure have to be gathered into a programming structure like a package, a module, a class, etc. Then, some powerful (multiple) inheritance mechanisms are required on these programming structures, to ease the programming task. The semantics of multiple inheritance should be clearly stated and if possible, formally studied. Indeed, on the proof side, a lot of lemmas rest upon mathematical inheritance between algebraic structures. This mathematical inheritance is reflected by programming inheritance. Even if we do not want to prove the internal mechanisms of inheritance of the host language, we must be confident of their correctness.
The way of packaging being chosen, we have to deal with overloading. Often, names are qualified by the name of the package, offering a first step of resolution of overloading. For example, the distinction between the operations $+_A$ and $+_B$ is achieved by qualification of names, which are $\mathbb{A}\text{plus}$ et $\mathbb{A}\text{plus}$. Thus, overloading has to be considered only inside a given algebraic structure, where we forbid it. This is the case for external ($*_B$ and $*_B$) and internal ($*_B$) multiplications of the $\mathbb{A}$-algebra $B$ which serve together to define $B$. Thus we do not follow the mathematical habit and we require different names. Indeed, it is rarely the case that such an overloading is needed. For the proof side, this is a sound choice as the properties of these operations differ and as explicit conversion must be done, with the help of the ring morphism $f$.

2.3 Coding mathematical specifications

We pursue our example, with a language of classes, which correspond to our implementation. How to assert that a particular set $\mathbb{A}$ is a group with carrier $\tau_\mathbb{A}$? As an example, we describe a stand-alone class (or a package or a module) additive\_group. It is parameterized by the type $\tau_\mathbb{A}$, which exports some methods. We use an Ocaml like syntax. Calling a method $m$ of an object $\text{obj}$ is written $\text{obj}.m$. The carrier of the group $\mathbb{A}$ appears only as a type parameter of the class.

```
class virtual [\tau_\mathbb{A}] additive\_group =
  object($\mathbb{A}$)
    method $-$ : $\tau_\mathbb{A}$ -> $\tau_\mathbb{A}$ -> bool
    method $+$ : $\tau_\mathbb{A}$ -> $\tau_\mathbb{A}$ -> $\tau_\mathbb{A}$
    method $0$ : $\tau_\mathbb{A}$
    method opposite : $\tau_\mathbb{A}$ -> $\tau_\mathbb{A}$
    method $a - b = a + (-b)$
    ...
  end
```

The operation $+_A$ is encoded by the operation $A # +$. The $\mathbb{A}$ appearing in $\text{object}(A)$ denotes the current object, like the self of several object languages. Operations are described by a curried type. This is only a matter of taste.

To describe rings, we do not explicitly rewrite all the methods but we use inheritance$^4$. Note that the type parameter denoting the carrier of the ring serves as an actual parameter for the inheritance declarations. This is a sound encoding of the mathematical specification. Indeed, as they are built upon the same set, the additive group and the multiplicative monoid share equality of this set. This sharing might be extended to the monoid structure, if addition and multiplication could be seen as two instances of a same parameter. This is possible in a proof language like Coq but this is difficult to handle in programming languages, due either to decidability of typing or to the impossibility of changing method names. Thus, we consider separately addition and multiplication, defining two separate notions of monoid.

```
class virtual [\tau_\mathbb{A}] ring =
  object($\mathbb{A}$)
    inherit [\tau_\mathbb{A}] additive\_group
    inherit [\tau_\mathbb{A}] multiplicative\_monoid
    ...
  end
```

$^4$ Indeed, additive\_group inherits from additive\_monoid.
Some hints for polynomials in the FOC project

A \( A \)-module is parameterized by the ring \( A \). To express such parameterized structures, we use value parameters and constraints on them. For example, the constraint on \( A \) expresses that \( A \) should be a ring. This feature, which is offered only by a few programming languages, has to be semantically well-understood. It permits to express some parts of the mathematical specification, directly in the programming language. In the following code, the parameter \( A \) represents an Ocaml object denoting the underlying ring, \( \tau_A \) represents its carrier. As an Ocaml object, \( A \) has a type denoted by \( \sigma_A \), which is constrained to be compatible with the class \([\tau_A] \) ring. This property is checked by the Ocaml type-checker. \( \tau_M \) represents the carrier of the module itself.

```ocaml
class virtual [\( \sigma_A, \tau_A, \tau_M \)] left_module (\( A : \sigma_A \)) =
object(M)
  constraint \( \sigma_A = (\tau_A)\#\text{ring} \)
  inherit \([\tau_M]\) additive_group
  method \(*_l : \tau_A \rightarrow \tau_M \rightarrow \tau_M \)
...
end
```

We can now proceed with the definition of algebras over commutative rings by specifying the ring morphism \( f \), here denoted by \textit{coerce}. As before, the parameter \( B \) (in \texttt{object(B)}) denotes the structure being specified, its carrier being denoted by the type parameter \( \tau_b \). The ring \( A \) is denoted by the value parameter \( A \), its carrier by \( \tau_A \) and its properties by \( \sigma_A \). These properties are given by the constraint \texttt{constraint \( \sigma_A = (\tau_A)\#\text{commutative_ring} \)}.

```ocaml
class virtual [\( \sigma_A, \tau_A, \tau_B \)] algebra (\( A : \sigma_A \)) =
object(B)
  constraint \( \sigma_A = (\tau_A)\#\text{commutative_ring} \)
  inherit \([\sigma_A,\tau_A,\tau_B]\) module (\( A \))
  method \textit{coerce} : \tau_A \rightarrow \tau_B
  method \( a \star b = (B\#\text{coerce } a) \star b \)
...
end
```

Note that the external multiplication is defined exactly as it is in the mathematical specification of an \( A \)-algebra.

We end this section by a few words on equality. Some algorithms need to check equality between two elements or check elements to 0. Thus, equality, or at least the considered implementation, should be effective, that is decidable. In the current state of our project, we focus on polynomial arithmetics which heavily use zero checks and we have made this rather strong hypothesis on equality.

### 2.4 Species of \textit{Foc}

The previous presentation uses several views of mathematical structures, which are considered sometimes as algebraic structures as in any algebra book, sometimes as mathematical specifications, sometimes as (rather abstract) implementations, sometimes as arguments for proofs, etc. We call \textit{species} our own notion of algebraic structures, seen as mathematical data submitted to effective coding and proving. For example, there is a species (called informally here the 
rooting\ species of monoids) which corresponds only to the mathematical definition of a monoid. Then, this species serves to build a new species, which defines natural numbers as a monoid, leaving the precise choice of the implementation of the carrier. This last species is used to build the species of, say, big natural numbers, whose representation is given by the library GMP and whose
operations are described by functions submitted to invariant properties. Thus, species
have a mathematical counterpart, but also a programming and a proving ones. We have
to embrace all these aspects at a time, any decision about one of these parts having
possibly heavy consequences on the other ones.

The encoding of species into Ocaml is presented in this paper. The specification of
species has been heavily studied in S. Boulmé’s thesis[2, 3], by coding it in Coq and
giving a categorical model à la Cartmell[19]. The introduction of a new species has been
decomposed into atomic steps, corresponding to atomic stages of proof correctness. A
concrete syntax for species is currently under design[15]. This syntax is compiled into
Ocaml code, which is very close to the code already written by hand. This syntax will
serve also to build statements and proofs, which are to be verified by the Coq verifier
(at the moment, proofs are done directly in Coq).

A species is defined by an ordered bunch of components, described rather informally
in a polymorphic typed framework. The first component is the carrier of the species,
the simplest one being a type variable $\tau$. For example, the parameter $\tau_1$ represents the
carrier of $\textsf{additive\_group}$. As we shall see below, $\tau$ may progressively be instantiated
by a type expression still containing other type variables or by an explicit data type.
Thus, a first way of creating a new species is a carrier instantiation.

The primitive components of a species are named and described by their prototype,
written as a type expression possibly depending on $\tau$ (or by a logical statement de-
pend on $\tau$ for components recording properties). This is the case of the $+$ method
and of the opposite method of $\textsf{additive\_group}$.

A given species can also have derived components which receive, beside a name and
a prototype, an implementation build upon the primitive components (and functional-
ities supposed available over $\tau$). The method $(\ldots)$ of $\textsf{additive\_group}$ is defined using
$+$ and opposite.

A second way to create new species is to extend a given species by adding primitive
or derived components. For instance $\textsf{ring}$ is an extension of the species $\textsf{additive\_group}$.
Sometimes, an extension adds only new properties: an abelian group has the same
operations than a group but has new properties.

Now, a primitive component of a species can receive an implementation, defining a
new species by a way usually called a refinement (so to extension of the specification,
only a step to approach a full implementation). The code has only to meet the declared
properties of the component. Thus the refinements of a species share names, prototypes,
some properties and some definitions. This will be illustrated below.

A component, say $c$, of a given species $\mathcal{S}_1$ may be redefined, leading to a new species
$\mathcal{S}_2$. As in the previous case, the new code has to meet the declared properties of the
component in $\mathcal{S}_1$. Moreover, as re-definitions of a species share also names, prototypes,
and some properties, if some of these properties in $\mathcal{S}_1$ rely upon the code of $c$, they have
to be reproved. For example, the method $*$ of $\textsf{algebra}$ redefines that of $\textsf{left\_module}$.

Whenever every primitive component of a species has a definition, this species can
only be extended by derived components. We will call collection such a species if we
do not want to extend it anymore. A collection gives a complete implementation of a
mathematical set, ready for users. It is created by applying an encapsulation mech-
anism to the given species, to forbid direct use of data representation so for modularity
purposes. Here, we use this notion only for the following definition.

Species can receive parameters as long as those are collections or entities. Thus, a
parameterized species is a kind of “function” taking collections or entities and returning
a species. For instance, $\textsf{algebra}$ is parameterized by the ring $\mathcal{A}$. As we will see below,
previous operations on species also apply to parameterized species.
A species \( S_1 \) can be converted into a species \( S_2 \) by establishing a correspondence between the primitive components of \( S_2 \) and some components of \( S_1 \), ensuring the same properties. A species can always be restricted to another species of which it is an extension. Namely a field can always be provided where a ring is wanted. This conversion requires a rather complex handling of dependencies, which is not described here.

3 Polynomials

In this section we will carry out the example of the implementation of polynomials. We will first describe univariate polynomials and then proceed toward multivariate polynomials.

3.1 The species of univariate polynomials

Depending of authors, there are several mathematical definitions of polynomials, as the solution of a universal problem, as an almost null sequence of coefficients, etc. We choose to have two different primitive notions of polynomials, according to their arrangement of variables, which correspond to two different species rooting the different implementations of polynomials.

Let \( A \) be a ring, \( A[X] \) is the ring of univariate polynomials with coefficients on \( A \). It is a commutative \( A \)-algebra, together with some basic primitives such as the degree, the leading coefficient. Usually the degree of a polynomial \( P \) of \( A[X] \) is a non-negative integer. In fact, the set \( D \) of degrees is simply required to be a regular additive monoid (a monoid with simplification : \( \forall x, y, z \in D, x + z = y + z \implies x = y \)). This monoid should have a total well-founded ordering \( \leq \) compatible with addition : \( \forall x, y, z \in D, x \leq y \implies x + z \leq y + z \) with 0 as minimal element. This guarantees correct use of the common rules that \( X^d \ast X^{d'} = X^{d+d'} \) and \( X^0 = 1_{A[X]} \).

We give first the coding of the species describing monomial orderings, still using Ocaml syntax.

```ocaml
class virtual ['TD] monomial_ordering =
  object (D)
    inherit ['TD] regular_additive_monoid
    inherit ['TD] total_ordering
  end
```

Notice that the compatibility of \( < \) upon + is not explicit in the previous code nor the fact that 0 is minimal for \( < \). Indeed, such invariant properties cannot be directly reflected in the running code. But, in our user language (currently under design), such properties need to be stated and proved.

Then, the coding of the (rooting) species of univariate polynomials is written :

```ocaml
class ['SA, TA, SD, TA, TP]
  abstract univariate_polynomials (A : 'SA, D : 'SD) =
    object (A[X])
      constraint 'SA = (TA)#commutative_ring
      constraint 'SD = (TD)#monomial_ordering
      inherit [(TP)] commutative_ring
      inherit ['SA, TA, TP] algebra (A)
      method |. | : TP -> TD
      ...
    end
```
3.2 Implementing the sparse representation

Usually in computer algebra univariate polynomials are encoded using either a dense or a sparse representation.

Dense polynomials are encoded using vectors of elements of $A$, indexed by elements of $N$. We will not give a formalization of these dense polynomials since to obtain an efficient implementation one is lead to have contiguous memory cells for the elements, destructive operations over the vectors and very often require user explicit pointer manipulation. We do not want to tackle now the formalization of such operations. However, this can be done using matlab21, for example and some of these features have been encoded in Coq by Filliâtre10.

We focus on sparse polynomials where a polynomial is encoded using a list of “monomials” which are pairs of non null elements of $A$ and elements of $N$. One can give an inductive definition of the sparse representation as follows:

\[
\begin{align*}
 P_0 & = \{0, a \} & |a| \neq 0_A \\
 P_d & = \bigcup_{d' \leq d} P_d \bigcup \{(d, a, p) \mid d \in N, a \in A, p \in P_{d'}, a \neq 0_A \}
\end{align*}
\]

Here $P_d$ is the set of sparse polynomials of degree at most $d$, $A[X]$, the set of sparse polynomials is the inductive limit of all possible $P_d$. This limit exists because the ordering on $N$ is well founded. Note that monomials are ordered by decreasing order of degree. If this can be reflected in the implementation, this enables to access both the degree and the leading coefficient of a polynomial in constant time.

We now need to implement this abstract recursive data structure, which can be thought as a list of pairs. There are several ways of implementing lists: arrays, pointers, classes, or recursive types. As already said, we reject arrays and pointers. We reject also any too abstract view of lists, either given by abstract data types (or modules) or given by classes, which offer only access to the head or the queue of the list. Indeed, the representation of the entities is intended to model the effective data and operations on this data must use as far as possible the internal definition of the data. Thus, we choose to encode the representation of sparse polynomials by a recursive concrete type:

\[
\text{type } (\tau_A, \tau_N)_{A[X]} = \\
| \text{ Z } | \\
| \text{ N of } (\tau_N \times \tau_A) \times (\tau_A, \tau_N)_{A[X]} |
\]

Then, implementations of functions, like $| P |$, can be defined by pattern-matching, which is usually an efficient way to access parts (like $d$ in $| P |$) of data. Moreover, pattern-matching eases correctness proofs. Without it, $d$ would be obtained by a more complicated code looking like “if list is not empty then (first (head list))”.

\[
\text{class } [\tau_A, \sigma_A, \tau_D, \sigma_D] \\
\text{sparse_univariate_polynomials } (A:\sigma_A, D:\sigma_D) = \\
\text{object } (A[X]) \\
\text{constraint } \sigma_A = (\tau_A)\#\text{commutative_ring} \\
\text{constraint } \sigma_D = (\tau_D)\#\text{monomial_ordering} \\
\text{inherit } [\sigma_A, \tau_A, \sigma_D, \tau_D, (\tau_A, \tau_D)_{A[X]}] \\
\text{abstract_univariate_polynomials } (A, D) \\
\text{method } | P | = \text{match } P \text{ with} \\
| \text{ Z } -> 0_D \\
| \text{ N } ((d, c), r) -> d \\
\ldots
\]

Our implementation of the degree function assumes that the first monomial of a non null polynomial is the one with highest degree. This cannot be reflected directly in the Ocaml type and has to be maintained as an invariant of the implementation.
Here, we have defined a new species, \texttt{sparse_univariate_polynomials}, from the one called above \texttt{abstract_univariate_polynomials}. We use the inheritance mechanism. We have made a carrier instantiation of \( \tau_P \) with the more defined expression \((\tau_A, \tau_D)\}_{\tau_A[X]}\), still containing type variables. We have implemented the function \( \lfloor P \rfloor \), doing a step of refinement. Thus, we have built this new species by a composition of primitive operations on species.

This new species is parameterized by the ring of coefficients and the set of degrees, which are given as actual parameters to \texttt{abstract_univariate_polynomials}. Therefore, the constraints on them are automatically checked by Ocaml and they could remain implicit. We choose to make them explicit as they are parts of the mathematical specification of the sparse polynomials.

### 3.3 Multivariate Polynomials

We do not describe here the (rooting) species of multivariate polynomials. We explain only our choice for the representation of these polynomials.

To represent polynomials in several variables, say \(X \) and \(Y\), we can use either a distributed representation, either a recursive representation.

Let \(N_1\) be the set of degrees in \(X\) and \(N_2\) be the set of degrees in \(Y\). The distributed representation is parameterized by a well founded ordering on \(N_1 \times N_2\). This ordering can be the lexicographical ordering, the total degree ordering, etc. We use the latter sparse representation with the set \(D = N_1 \times N_2\) as the set of degrees of the polynomials. We thus see that distributed representations are a generalization of sparse univariate encoding.

For the recursive representation of polynomials in \(X\) and \(Y\), we use a precedence between \(X\) and \(Y\), say \(Y\) greater than \(X\). Thus, we view polynomials in \(X\) and \(Y\) as univariate polynomials in \(Y\) whose coefficients are polynomials in \(X\). We distinguish the case of constant polynomials, which are considered as constant polynomials in any variable. The recursive representation of polynomials in \(X\) and \(Y\) is defined as follows.

\[
\begin{align*}
P_0 &= A \\
P_1^X &= P_0 \cup \{(1, p) | p \in P_0, p \neq 0_A\} \\
P_{i+1}^X &= P_i^X \cup \{((i + 1, p), q) | q \in P_i^X, p \neq 0_A\} \\
P_X &= \bigcup_i P_X^i \\
P_{X,Y}^i &= P_X \cup \{(1, p) | p \in P_X, \deg p > 0_N\} \\
P_{j+1}^{X,Y} &= P_j^{X,Y} \cup \{((j + 1, p), q) | q \in P_j^{X,Y}, p \in P_X, p \neq 0_A\} \\
P^{X,Y} &= \bigcup_j P_j^{X,Y}
\end{align*}
\]

The set \(P^{X,Y}\) is the set of all polynomials in \(X\) and \(Y\). It is well-defined as the inductive limit of the \(P_i^{X,Y}\). Note that precedence is used to construct the fix-point step by step.

To obtain the set \(P\) of all polynomials, we need to choose a precedence, which is a total well-founded ordering on variables. Let \((X_i)\) be the ordered set of variables. Then, \(P = \bigcup_i P^{X_1,\ldots, X_i}\) and the definition of \(P^{X_1,\ldots, X_i}\) is the straightforward generalization of the one of \(P^{X,Y}\). Here is the corresponding type for the carrier, in a Ocaml like syntax:

\[
\begin{align*}
\text{type } (\tau_A, \tau_N, \tau_V)_{\tau_A[X_1\ldots X_n]} = \\
| \ G \ of \ \tau_A \\
| \ P \ of \ \tau_V \ast ((\tau_A, \tau_N, \tau_V)_{\tau_A[X_1\ldots X_n]}; \ \tau_V)_{\tau_A[X]}
\end{align*}
\]
Here $(\tau_A[X_1,\ldots,X_n], \tau_V)$ is the type for univariate polynomials with coefficients over $\tau_A[X_1,\ldots,X_n]$ using $\tau_V$ to represent the degrees and $\tau_V$ for indexing the variables.

For example, the polynomial $(5X^7 + 3X)^2 + 2$ is coded as follows:

\[ P(Y, \mathbb{N}(\{8, P(X, \mathbb{N}((7, \mathbb{G} 5), \mathbb{N}((1, \mathbb{G} 3), \mathbb{Z}))))), \mathbb{N}((0, \mathbb{G} 2), \mathbb{Z})) \]

Again, invariants given by the orderings are lost in this definition. We cannot deduce from the type, that if a polynomial is non constant, it is a univariate polynomial in its main variable.

To end this example, we will define $P$ as a commutative $A$-algebra with carrier $(\tau_A, \tau_N, \tau_V)$ and the last type is abbreviated into $\tau_P$, using the keyword as.

To implement the operations of $A[X_1,\ldots,X_n]$ we need to view it as the set of univariate polynomials $A[X_1,\ldots,X_{n-1}],X_n]$. Note that the representation of $A[X_1,\ldots,X_{n-1}]$ has its type also described by $P$. Therefore, we are led to call operations of $P[X]$.

We have already an implementation of sparse univariate polynomials, given by the \texttt{sparse_univariate_polynomials} class. Therefore, we reuse it.

The following code is very close to the running one. It expresses a lot of mathematical but also implementation dependencies. We have omitted some details to ease the comprehension. But, as the reader can see, this code is not so simple.

```ocaml
class [\sigma_A, \tau_A, \sigma_D, \tau_D, \sigma_V, \tau_V] recursive sparse_multivariate_polynomials
  (A: \sigma_A, D: \sigma_D, V: \sigma_V)
object (P: \sigma_P)
  constraint \sigma_A = (\tau_A)\#\texttt{commutative\_ring}
  constraint \sigma_D = (\tau_D)\#\texttt{monomial\_ordering}
  constraint \sigma_V = (\tau_V)\#\texttt{ordered\_set}
  inherit [(\tau_A, \tau_N, \tau_V)\tau_A[X_1,\ldots,X_n] as \tau_P] \texttt{commutative\_ring}
  inherit [\sigma_A, \tau_A, \tau_P] \texttt{algebra}(A)
method P, [X] = \texttt{new} \texttt{sparse\_univariate\_polynomials}(P, D)
  ...
end
```

The method $P, [X]$ enables to call operations from the ring $P[X]$ which are implemented in the class \texttt{sparse\_univariate\_polynomials}. This method is correctly typed. Indeed, $P$ is an Ocaml object with type $\sigma_P$, this type is compatible with \texttt{commutative\_ring} since the class of $P$ inherits from it.

## 4 Conclusion

In this paper, we have tried to give some insights on the building of the Foc library, following the implementation of polynomials as a running example. No algorithm on polynomials is presented here, due to the lack of place. But we can say that the subresultant algorithm was encoded using the recursive representation, without any difficulty. Running it on classical benchmarks shows that it meets our requirements on time and memory efficiency. These benchmarks are available at \url{www-calfor.lip6.fr/~foc}.

Designing the Foc library, we were always concerned by its emerging proof counterpart. This led us to introduce the notion of species, which gathers together different views of effective mathematics. The specification of the species and the properties of operations manipulating them are formally studied in [2]. This paper may be viewed as a presentation of the implementation of species, the motivations of our choices and their justification.

The Foc library implements general computer algebra notions without the help of a dedicated computer algebra language. Using a general purpose language has some
advantages, among them to have libraries for data structures (lists, trees, graphs, etc.), for input-output in different formats, for interoperability with specialized libraries such as GMP. The major advantage is that it avoids writing a full compiler, a task which needs the skills of specialists. Moreover, a general purpose language is tested every day by its users. This gives a guaranty on its maintainability (correction of bugs and extensions).

However, using such a general language has several drawbacks. The first one is that we have to be confident on its semantic foundations. These ones have to be exposed and formally studied as long as possible. This is the case for Ocaml, which is based upon research[14, 16, 17] on types, modules, classes, etc. Thus, we can claim that this drawback is only a minor one. Moreover, any semantic flaw is eventually emerging, due to intensive uses of this language.

A more serious drawback is that a general language offers a lot of possibilities to encode algorithms, which are not guaranteed to fit our requirements. The first species were implemented by a very limited team, with permanent code review. But, we hope to have some helps to describe species for other mathematical domains and we need to elaborate a programming discipline. It makes a restricted use of object oriented features but uses the full power of the class sub-language. We tried to give a flavor on it with the examples (choice for the representation of the carrier, binary operations keeping their two arguments, constraint expressing mathematical facts, redefinition and late binding, etc.). This discipline is natural enough to be followed by some undergraduate students adding species (fractions, matrix) without true difficulties. But we think that we cannot rely only on willingness. Therefore, a concrete syntax has been defined for species. It disguises Ocaml syntax for declarations of types, classes and offers only restrictive constructions to write functions (no references for example), allowing however all the ones used in code written by hand like late binding. Therefore, this is not surprising that the parser producing Ocaml sources generates code very close to the code written by hand. The concrete syntax serves also to build statements, submitted to proofs. Thus, the implementation of species can be certified at the source level. We are aware that this does not give full guarantees but this is not realistic to attempt certifying executable object-code.

References

Incorporating Decision Procedures in Implicit Induction

Alessandro Armando
d, Michaël Rusinowitch
e, and Sorin Strătulat

1 DISS – Università di Genova, Viale Causa 13 – 16145 Genova – Italia
2 LORIA-INRIA, 615, rue du Jardin Botanique – 54602 Villers-lès-Nancy – France

Abstract In the last decades the automation of reasoning by mathematical induction has been thoroughly investigated and several powerful techniques and heuristics have been put forward. However, when applied to proof obligations arising in practical applications, the level of automation achieved by existing induction provers is still unsatisfactory. As shown by Boyer and Moore, a higher level of automation can be achieved by the incorporation of decision procedures into induction provers. Yet in Boyer and Moore’s approach the role of the decision procedure is confined to the simplification engine and this limits the possible usage of the decision procedure by the prover. In this paper we present an extension to Boyer and Moore’s integration schema that enables the decision procedure to use suitably selected instances of the induction hypotheses. The induction proof method we consider is based on and combines Cover Set Induction and Constraint Contextual Rewriting and has been implemented in the SPIKE prover. Computer experiments on non-trivial verification problems give evidence of the effectiveness of our approach: the proof of the MJRTY algorithm does not need anymore user-defined tactics as it is the case with STeP and Nuprl; moreover, in the proof of an ABR conformance algorithm, many of the about 80 user-defined lemmas require specific tactics with PVS whereas more than half of them are relieved automatically by our extended system.

1 Introduction

In the last decades the automation of reasoning by mathematical induction has been thoroughly investigated and several powerful techniques and heuristics have been put forward. However, when applied to proof obligations arising in practical applications, the level of automation achieved by existing induction provers is still unsatisfactory. As shown by Boyer and Moore [BM85], a higher level of automation can be achieved by the incorporation of decision procedures into induction provers. Yet in Boyer and Moore’s approach the role of the decision procedure is confined to the simplification engine and this limits the possible usage of the decision procedure by the prover.

In this paper we present an extension to Boyer and Moore’s integration schema that enables the decision procedure to use suitably selected instances of the induction hypotheses. The approach we propose is based on and combines Cover Set Induction and Constraint Contextual Rewriting. Cover Set Induction [BR95b] is a powerful automated reasoning technique for reasoning about inductively defined objects which combines the advantages of explicit induction and proof by consistency. Constraint
Contextual Rewriting [AR98, AR00], CCR or CCR(X)\(^1\) for short, is an abstract integration schema between rewriting and decision procedures. CCR(X) generalizes contextual rewriting [ZR85, Zha95] by allowing the available decision procedure to access and manipulate the rewriting context. One of the key features of CCR is the ability to augment the state of the decision procedure with facts encoding properties of symbols which are uninterpreted for the decision procedure. (As shown in [BM85], this feature is crucial to the effectiveness of the integration.) A key contribution of this paper is the extension of CCR so to enable the use of the induction hypotheses (other than the available definitions and lemmas) during the augmentation of the state of the decision procedure.

The extended induction proof method presented in this paper has been implemented in the SPIKE prover [BR95a]. Computer experiments on non-trivial verification problems give evidence of the effectiveness of our approach: the proof of the MJRTY algorithm [BM91] does not need anymore user-defined tactics or lengthy interactions as it is the case Nuprl [Jac94] and STeP [B’95]; moreover, in the proof of an ABR conformance algorithm, many of the about 80 user-defined lemmas require specific tactics with PVS [ORS92], whereas more than half of them are relieved automatically by our extended system.

Structure of the paper. In Section 2 we introduce the concept of reasoning specialist and specify the associated interface functionalities. CCR is defined in Section 3 and clause simplification is defined in Section 4. Our extended induction method is then presented in Section 5. A reasoning specialist for the union of the quantifier-free theory of equality and quantifier-free Presburger arithmetics is described in Section 6. An excerpt of the proof of the soundness of MJRTY is finally discussed in Section 7.

Preliminaries. By \(\Sigma\) (possibly subscripted) we denote finite sets of function symbols (with their arity). \(V\) (possibly subscribed) denotes a finite set of variables. \(\tau(\Sigma, V)\) is the set of terms built out of \(\Sigma\) and \(V\) in the usual way. \(\tau(\Sigma)\) abbreviates \(\tau(\Sigma, \emptyset)\), i.e. the set of ground terms. We assume the usual conceptual machinery (e.g. the notion of substitution, the definition of position of a sub-expression) as given, e.g., in [DJ90]. A \((\Sigma, V)\)-equation is an expression of the form \(t_1 = t_2\) where \(t_1, t_2 \in \tau(\Sigma, V)\). \((\Sigma, V)\)-formulae are built in the usual way using the standard logical connectives (i.e. \(\neg, \land, \lor, \Rightarrow, \Leftrightarrow\). A \((\Sigma, V)\)-literal is either a \((\Sigma, V)\)-equation or a negated \((\Sigma, V)\)-equation. We write \((\Sigma, V)\)-equation (-literal) instead of \((\Sigma, \emptyset)\)-atom (-literal, resp.). A \((\Sigma, V)\)-clause is a disjunction of literals which we italicize as finite set of \((\Sigma, V)\)-litersals. We denote by \textbf{true} any tautology of the form \(t \equiv t\), for any \(t \in \tau(\Sigma, V)\). If \(a\) is an atom, then \(\overline{a}\) abbreviates \(\neg a\); and \(\overline{\bigwedge S}\) stands for any conjunction of the literals in \(S\).

\(<\) is a reduction ordering over the \((\Sigma_j, V)\)-expressions (i.e. a well-founded relation over the \((\Sigma_j, V)\)-expressions closed under substitution and replacement) containing the sub-expression relation. We also assume that \textbf{true} \(<\) \(<\) \textbf{false} for all equations \(e \neq \textbf{false}\). Given a congruence relation \(\approx\) on terms that is stable (i.e., \(s\sigma \approx s\sigma'\) if \(s \approx s'\)) and compatible with \(<\) (i.e., \(s' < t'\) if \(s < t\), \(s \approx s'\), and \(t \approx t'\)) we define \(\leq\) as \(<\cup \approx\). \(\ll\) is the multiset extension of \(<\). A conditional equation \(\lambda_{\leq 1} \sigma a_i = b_i \Rightarrow l \rightarrow r\) if, for each substitution \(\sigma\), \(\{\sigma\} \cup (\cup_{\leq 1} \{a,\sigma, b, \sigma\}) \ll \{l\sigma\}\). A conditional rewrite system is a set of conditional rewrite rules. Given a conditional rewrite system \(R\) obtained from the orientation of

\(^1\) The notation CCR(X) (by analogy with the CLP(X) notation used to denote the Constraint Logic Programming paradigm [JL87]) is used to stress the independence of CCR(X) from the theory decided by the decision procedure.
2 The Reasoning Specialist

A reasoning specialist is a state-based procedure whose states (called constraint stores) are finite sets of $\Sigma_i$-literals represented in some internal form and whose functionalities are abstractly characterized in the following way.

**Initialization of the Constraint Store.** The first functionality we consider is the relation $\text{cs-init}(S)$ which characterizes the "empty" constraint stores. $\text{cs-init}(S)$ is required to be a decidable relation such that $\text{cs-init}(S)$ holds only if $S$ is $T_c$-valid.

**Detection of Unsatisfiability.** $\text{cs-unsat}(S)$ characterizes a set of $T_c$-unsatisfiable constraint stores $S$ whose $T_c$-unsatisfiability can be checked by means of a computationally inexpensive syntactic check. We require that $\text{cs-unsat}(S)$ is decidable and that $\text{cs-unsat}(S)$ implies the $T_c$-unsatisfiability of $S$.

**Constraint Store Simplification.** The main functionality of the reasoning specialist is a transition relation over constraint stores, $S \xrightarrow{\text{cs-simp}} S'$, which models the activity of adding a finite set of $\Sigma_i$-literals $P$ to $S$ yielding a new constraint store $S'$. For soundness we require that $T_c \models \bigwedge(P \cup S) \Rightarrow \bigwedge S'$ whenever $S \xrightarrow{\text{cs-simp}} S'$.

**Augmentation.** Let $P$ be a finite set of literals. It is a trivial consequence of the above definitions the fact that if $S_0$ and $S$ are constraint stores such that $\text{cs-init}(S_0)$, $S_0 \xrightarrow{\text{cs-simp}} S$, and $\text{cs-unsat}(S)$ then $P$ is $T_c$-unsatisfiable. This observation shows that the functionalities we have presented so far allow us to check the $T_c$-unsatisfiability of any given set of literals $P$. Unfortunately in most cases $P$ (and hence $S$) is $T_j$-unsatisfiable but not $T_c$-unsatisfiable. When this is the case, the $T_c$-unsatisfiability of $S$ cannot possibly be detected by the reasoning specialist. The occurrence in $S$ of (function) symbols interpreted in $T_j$ but not in $T_c$ is the main cause of the problem. The key idea of augmentation is to extend $S$ with $T_j$-valid facts, thereby informing the reasoning specialist about properties of function symbols it is otherwise not aware of. By adding $T_j$-valid facts to the rewriting context, the heuristics aims at generating a $T_j$-equivalent but $T_c$-unsatisfiable context whose $T_c$-unsatisfiability can therefore be detected by the reasoning specialist. The selection of suitable $T_j$-valid facts is done by looking up $R$ or $H$ which contains the available induction hypotheses. To model augmentation we define a new relation, $S \xrightarrow{\text{cs-extend}} S'$ as the smallest transitive relation (i.e. such that
3 Constraint Contextual Rewriting

Constraint Contextual Rewriting is modeled by the relation $E \overset{cc}{\leftarrow_{R,H:P}} E'$ which is defined to be the smallest transitive relation such that

**Entailment Check:**

\[ e \overset{cc}{\leftarrow_{R,H:P}} \text{true} \]

if $e$ is a \( \Sigma\)-literal, \( e \neq \text{true} \), \( S \overset{cc}{\leftarrow_{R,H:P}} S' \) and \( \text{cs-unsat}(S') \)

**Conditional Rewriting:**

\[ c[l] \overset{cc}{\leftarrow_{R,H:P}} e[l] \]

if \( (Q \Rightarrow l = r) \in R \) or \( (Q \Rightarrow l \Rightarrow r) \in H \), and \( q \sigma \overset{cc}{\leftarrow_{R,H:P}} \text{true} \) for all \( q \in Q \).

**Theorem 1 (Soundness of CCR).**

1. If \( S \overset{cc}{\leftarrow_{R,H:P}} S' \) then \( R, H, P, S \models_{ini} \bigwedge S' \);
2. If \( e \overset{cc}{\leftarrow_{R,H:P}} e' \) then \( R, H, S \models_{ini} (e \sim e') \) and \( e' \prec e \);

**Proof.** Since the definitions of \( \overset{cc}{\leftarrow_{R,H:P}} \) are mutually dependent, we prove facts 1 and 2 by mutual induction. Let us consider a calculus comprising the rules cs-simp, augment, entailment check and CCR, and let us consider any sensible definition of derivation in such a combined/hybrid calculus. We reason by induction on the depth of the derivations. The base case (i.e. the number of occurrences of \( \overset{cc}{\leftarrow_{R,H:P}} \) and \( \overset{cc}{\leftarrow_{R,H:P}} \)) in the derivation is 1 amounts to proving the following case:

- \( S \overset{cc}{\leftarrow_{R,H:P}} S' \) results from the application of cs-simp and therefore \( S' \) is such that \( S \overset{cc}{\leftarrow_{R,H:P}} S' \) and hence \( T_e \models \bigwedge (P \cup S) \Rightarrow S' \) holds. From this 1 readily follows.

In the step case we must prove that 1 and 2 hold for all derivations of depth \( k + 1 \) provided that they hold for all derivations of depth \( k \). In the step case we have the following cases to consider:

- \( S \overset{cc}{\leftarrow_{R,H:P}} S' \) results from the application of augment and therefore \( S' \) is such that \( S \overset{cc}{\leftarrow_{R,H:P}} S' \) where either \( (Q \Rightarrow c) \in R \) or \( (Q \Rightarrow c) \in H \) and \( q \sigma \overset{cc}{\leftarrow_{R,H:P}} \text{true} \) for all \( q \sigma \in Q \). From the induction hypothesis we know that \( R, S \models_{ini} q \sigma \) for all \( q \in Q \). From this and the fact that \( (Q \Rightarrow c) \in R \) or \( (Q \Rightarrow c) \in H \) it readily follows that \( R, H, S \models_{ini} c \sigma \). By induction hypothesis we also know that \( R, H, c, S \models_{ini} \bigwedge S' \) and therefore we can conclude that \( R, H, S \models_{ini} \bigwedge S' \) and hence 1.
- \( e \xrightarrow{\text{cs}} \mathcal{R}; S \xrightarrow{\text{R}} e' \) results from the application of Entailment Check and hence \( e \) is a \( \Sigma_c \)-literal and \( e' = \text{true} \). In this case we know that \( S \xrightarrow{\text{cs-exten}, \Sigma_c} S' \) and \( \text{cs-unsat}(S') \). By induction hypothesis we have \( R, \mathcal{H}, \pi, S \models_{\text{ini}} \wedge S' \) and from the \( T_i \)-unsatisfiability of \( S' \) it readily follows that \( R, \mathcal{H}, S \models_{\text{ini}} e \). It is immediate to see that \( \text{true} \prec e \) holds.

- \( e \xrightarrow{\text{cs}} \mathcal{R}; S \xrightarrow{\text{R}} e' \) results from the application of Conditional Rewriting and there exists a substitution \( \sigma \) and a clause \( Q \Rightarrow l \Rightarrow r \) such that \( l\sigma \) occurs in \( e \), i.e. \( e = e[l\sigma], e' = e[r\sigma], Q \prec \{l\sigma = r\sigma\}, r\sigma \prec l\sigma \) and either \((Q \Rightarrow l \Rightarrow r) \in R \) or \((Q \Rightarrow l \rightarrow r) \sigma \in \mathcal{H} \); moreover \( q\sigma \xrightarrow{\text{cs}} \text{true} \) for all \( q \in Q \). By induction hypothesis we know that \( R, \mathcal{H}, S \models_{\text{ini}} l\sigma = r\sigma \) holds for all \( q \in Q \). If \((Q \Rightarrow l \Rightarrow r) \in R \) we can conclude that \( R, \mathcal{H}, S \models_{\text{ini}} l\sigma = r\sigma \) and therefore \( R, \mathcal{H}, S \models_{\text{ini}} e[l\sigma] \sim e[r\sigma] \). Also, \( e[r\sigma] \prec e[l\sigma] \) holds since \( r\sigma \prec l\sigma \) and \( \prec \) is monotonic and stable. The case in which \((Q \Rightarrow l \rightarrow r) \sigma \in \mathcal{H} \) is proven along the same lines.

4 Clause Simplification

Clause simplification is modeled by the relation \( E \xrightarrow{\text{simple}, \mathcal{H}} E' \) which is defined to be the smallest transitive relation such that:

**DELETE:** \( \{C\} \cup E \xrightarrow{\text{simple}, \mathcal{H}} E \) if \( \text{true} \in C \)

**CCR:** \( \{\{p\} \cup C\} \cup E \xrightarrow{\text{simple}, \mathcal{H}} \{\{p'\} \cup C\} \cup E \) if \( \text{cs-init}(S_0), S_0 \xrightarrow{\text{cs-exten}, \Sigma_c} S \) and \( p \xrightarrow{\text{cs}} \mathcal{R}; S \xrightarrow{\text{R}} p' \)

**Theorem 2 (Soundness of Clause Simplification).** If \( E \xrightarrow{\text{simple}, \mathcal{H}} E' \) and \( \prec \) is monotonic, then for all clauses \( C \in E \) either \( \text{true} \in C \) or there exists a clause \( C' \in E' \) such that \( R, \mathcal{H} \models_{\text{ini}} (C \Leftrightarrow C') \) and \( C' \prec_c C \).

The proof of this result is straightforward and therefore it is omitted.

5 Cover Set Induction

Let \( R \) be a rewrite system derived from the orientation of a set of axioms \( A_x \). A term \( t \) is said to be inductively \( R \)-reducible (resp. \( R \)-irreducible) if, for each substitution \( \gamma \) mapping variables to \( R \)-irreducible terms, \( t\gamma \) is \( R \)-reducible (resp. \( R \)-irreducible). A cover set for a conditional rewrite system \( R, CS(R) \), is a finite set of \( R \)-irreducible terms such that for all ground \( R \)-irreducible term \( s \), there is a term \( t \in CS(R) \) and a ground substitution \( \sigma \) such that \( A_x \models t\sigma = s \). From a cover set we can build cover sets for clauses. A cover substitution for a clause \( C \) instantiates a particular subset of \( \text{Var}(C) \) (called induction variables) by terms obtained from \( CS(R) \) whose variables are replaced by fresh ones. We will denote by \( CS\Sigma(C) \) the set of all possible cover substitutions for the clause \( C \). Then, the set \( \{C\sigma \mid \sigma \in CS\Sigma(C)\} \) is a cover set for the clause \( C \).

The induction method we consider incrementally modifies two sets of clauses, \( (E, H) \), where \( E \) contains the conjectures to be checked and \( H \) contains clauses, previously in \( E \), that have been reduced. The method is modeled by means of the relation \( (E, H) \xrightarrow{\text{split}, \text{cs}} (E', H') \) which is defined to be the smallest transitive relation such that:
\[
\text{Generate: } (E \cup \{C\}, H) \xrightarrow{\text{ spike }} (E \cup \bigcup_{\sigma \in C \Sigma(C)} E_{\sigma}, H \cup \{C\}) \\
\text{if } \{C\} \xrightarrow{R_{\sigma}(E \cup H \cup \{C\}) \cap C \neq \emptyset} E_{\sigma} \text{ for } \sigma \in C \Sigma(C)
\]

\[
\text{Simplify: } (E \cup \{C\}, H) \xrightarrow{\text{ spike }} (E \cup E', H) \\
\text{if } \{C\} \xrightarrow{R_{\sigma}(E \cup H) \cap C \neq \emptyset} E'
\]

The above two rules synthesize a simplified version of the current inference system of SPIKE. The Generate inference rule computes the covering substitutions which are then applied to conjectures thereby generating special instances which are then simplified by rules, lemmas and the available induction hypotheses. The Simplify inference rule simplifies conjectures. The set of induction hypotheses are ad-hoc instances of the current set of \(E, \{C\}\) and \(H\) depending on the treated clause \(C\).

\(E_0\) is an inductive theorem w.r.t. \(Ax\) if there exists a finite derivation of the from \((E_0, \emptyset) \xrightarrow{\text{ spike }} \cdots \xrightarrow{\text{ spike }} (\emptyset, H)\). More in general, we say that \(E_0\) is an inductive theorem w.r.t. \(Ax\), in symbols \(Ax \vdash_{\text{ini}} E_0\), iff there exists a fair derivation \((E_0, \emptyset) \xrightarrow{\text{ spike }} (E_1, H_1) \xrightarrow{\text{ spike }} \cdots\), i.e. iff the set of persisting clauses \(\bigcup_{i \geq 0} \bigcap_{j \geq i} E_j\) is empty.

**Theorem 3 (Soundness of Cover Set Induction).** If \(Ax \vdash_{\text{ini}} E_0\) then \(Ax \models_{\text{ini}} E_0\).

**Proof (Adapted from [Bow97]).** Let us assume that \(Ax \vdash_{\text{ini}} E_0\) but \(Ax \not\models_{\text{ini}} E_0\). Since \(Ax \vdash_{\text{ini}} E_0\), then there exists a fair derivation \((E_0, \emptyset) \xrightarrow{\text{ spike }} (E_1, H_1) \xrightarrow{\text{ spike }} \cdots\). Let \(C \in \bigcup_{i} E_i\) be one of the last clauses in the derivation containing a minimal counterexample (w.r.t. \(\prec\)) from the set

\[CE = \{D\theta : D \in \bigcup_{i} E_i\text{ and }Ax \not\models_{\text{ini}} D\theta \text{ for some ground } R\text{-irreducible substitution } \theta\}\]

(Notice that \(CE\) is not empty since, by assumption, \(Ax \not\models_{\text{ini}} E_0\); moreover \(CE\) has a minimal element w.r.t. \(\prec\) since \(\prec\) is well-founded.) Since the derivation is fair, there must exist a pair \((E\{\tau\}, H), H\) in the derivation such that either Generate or Simplify apply to it. It suffices to show that either Generate or Simplify may affect \(C\), since this trivially contradicts the fairness assumption.

(1) Case: Generate. As \(\varphi\) is ground and \(R\)-irreducible, then there exists \(\sigma \in C \Sigma(C)\) and a ground substitution \(\tau\) such that \(\varphi = \sigma \tau\). From Theorem 2 it follows that there exists a clause \(C' \in \bigcup_{i} E_i\) such that \(R_{\tau}(E \cup H \cup \{C\}) \cap C \neq \emptyset\) \(\models_{\text{ini}} (C\varphi \equiv C'\tau)\) with \(C'\tau \prec C\varphi\).\footnote{Notice that \textbf{true} cannot possibly be in \(C\varphi\) as this would contradict the assumption that \(C\varphi \in CE\).} Notice that \(Ax \models_{\text{ini}} (E \cup H \cup \{C\}) \prec C\varphi\) must hold too (otherwise the minimality of \(C\varphi\) in \(CE\) would be contradicted) and hence \(Ax \models_{\text{ini}} (C\varphi \equiv C'\tau)\). Since \(Ax \not\models_{\text{ini}} C\varphi\), then also \(Ax \not\models_{\text{ini}} C'\tau\) which, together with \(C'\tau \prec C\varphi\), contradicts the minimality of \(C'\varphi\) in \(CE\).

(2) Case: Simplify. From Theorem 2 it follows that there exists a clause \(C' \in \bigcup_{i} E_i\) such that \(R_{\tau}(E \cup C\varphi \cup H) \models_{\text{ini}} (C\varphi \equiv C'\varphi)\) and \(C'\varphi \prec C\varphi\). This case is similar to the previous one with the additional proof obligation of showing that if a clause \(C_i\) from \(H\) is such that \(C_i\theta \models_{\text{ini}} C\varphi\) for some ground \(R\)-irreducible substitution \(\theta\), then \(Ax \models_{\text{ini}} C_i\theta\). Let us assume that \(Ax \not\models_{\text{ini}} C_i\theta\), then \(C_i\theta\) must also be minimal in \(CE\). But \(C_i\) can be put in \(H\) only by a previous application of Generate which is in contradiction with the previous case.

Notice that \textbf{true} cannot possibly be in \(C\varphi\) as this would contradict the assumption that \(C\varphi \in CE\).
6 A Reasoning Specialist for the union of the quantifier-free theory of Equality and quantifier-free Presburger Arithmetics

We present a reasoning specialist for the union of quantifier-free Presburger Arithmetic, $T_{\text{pc}}$, and the quantifier-free theory of equality, $T_{\text{eq}}$, obtained by combining a decision procedures for $T_{\text{pc}}$ and one for $T_{\text{eq}}$. The interface functionalities of the composite decision procedure are as described in Section 2 and we show that they comply with the requirements stated in the same section.

The decision procedure manipulates constraint stores of the form $(A \mid U \mid G \mid (P \cdot I))$ where $A$ is a set of literals, $U$ is a set of ground rewrite rules, $G$ is a set of ground equations and disequations, $L = (P \cdot I)$, where $P$ is a set of linear inequalities and $I$ is a set of equations entailed by $P$. $\text{cs-init}(A \mid U \mid G \mid (P \cdot I))$ holds iff all the fields are empty. $\text{cs-unsat}(A \mid U \mid G \mid (P \cdot I))$ holds whenever there exists either an impossible inequation in $P$ or a disequation of the form $a \neq b$ in $G$. $(A \mid U \mid G \mid (P \cdot I)) \xrightarrow{\text{cs-imp}} (A' \mid U' \mid G' \mid (P' \cdot I'))$ is defined by $(A \cup U \mid U \mid G \mid (P \cdot I)) \xrightarrow{\text{dataflow}} (A' \mid U' \mid G' \mid (P' \cdot I'))$.

$$\text{A2L:} (A \mid U \mid G \mid (P \cdot I)) \xrightarrow{\text{cs-imp}} (A' \mid U \mid G \mid (P' \cdot I' \cup I'))$$
if $A' = \{ a \in A : \text{linearize}(a) = 0 \}$ and
$(P', P'') = \text{arith}(\cup_{a \in A} \text{linearize}(a) \cup P)$

$$\text{A2G:} (A \mid U \mid G \mid (P \cdot I)) \xrightarrow{\text{cs-imp}} (A \setminus A' \mid U \mid G \cup A' \mid (P \cdot I))$$
if $A' = \{ a \in A : \text{linearize}(a) = 0 \}$

$$\text{G2U:} (A \mid U \mid G \mid (P \cdot I)) \xrightarrow{\text{cs-imp}} (A \mid U \cup E \mid \text{congr}(G) \mid (P \cdot I))$$
if for all $a \rightarrow b \in E$ we have that $a = b \in \text{congr}(G)$ and $b < a$

$$\text{12G:} (A \mid U \mid G \mid (P \cdot I)) \xrightarrow{\text{cs-imp}} (A \mid U \mid G \cup I \mid (P \cdot 0))$$

NORMALIZE: $(A \mid U \mid G \mid (P \cdot I)) \xrightarrow{\text{cs-imp}} (A \mid U \mid G' \mid (P' \cdot I'))$
if $P'$ and $I'$ are the results of normalizing $P$ and $I$ (resp.) with the rules in $U$

The effect of the application of the rules is that of moving information from the fields of the constraint store as depicted in Figure 1.

A2L initializes the component $P$ with linear inequalities resulting from the linearization of literals from $A$, then it applies $\text{arith}$ on the current set of linear inequalities from $P$. linearize is a function that maps literals into sets of linear inequalities, according to the mapping specified in Table 1. linearize returns the empty set if no transformation can be performed on the input literal (and in such a case the input literal is said to be non-linearizable). arith models the functionality of a (semi-)decision procedure for linear arithmetic based on the Fourier-Motzkin variable elimination method [LM92]. Its input is a set of linear inequalities and returns a set of inequalities and a set of entailed equations.

$^3$ From now on, by ground rewrite rule (resp. equation) we mean a rewrite rule (resp. equation) whose variables are forbidden to be instantiated.

$^4$ In the last column, the members of the equality $s = t$ must be terms of arithmetic sort. For brevity we omit the treatment of disequations.
The A2G rule initializes the $G$-component with the non-linearizable literals of $A$. The G2U rule transfers to $U$ all the new rewrite rules obtained after the application of \textit{congr} on $G$. \textit{congr} models the application of an algorithm for ground completion [HL78]. These equations are oriented in unconditional rewrite rules using the ordering $\prec$. The rewrite system is used to normalize the left and right hand sides of the disequations. The L2G rule transfers the implicit inequalities from $L$ to $G$. \texttt{NORMALIZE} normalizes the monomials in $P$ and the ground equations in $I$ by means of the rewrite rules in $U$.

**Theorem 4 (Soundness of the Reasoning Specialist).** Let \texttt{wfs}(S) be the set of literals stored in the constraint store $S$, then the following facts hold:

- If \texttt{cs-unsat}(S) then \texttt{wfs}(S) is $(T_a \cup T_{eq})$-unsatisfiable.
- If $S \xrightarrow{\text{imp}} S'$ then $T_a \cup T_{eq} \models \wedge \text{wfs}(S') \iff \wedge (A \cup \text{wfs}(S))$.

A proof of this result can be found in [Str00b].

## 7 Proving the Soundness of MJRTY

Given a multiset of elements as input, the MJRTY algorithm computes in an efficient way its majority element (if any), i.e. the element occurring more than the half of the multiset cardinality. The algorithm scans the elements in real time, without additional storage of elements for further operations, and eliminates the counting phase specific
to other similar (trivial) algorithms. MJRTY has been devised in 1980 by Boyer and Moore who have also proved its soundness by means of their prover NQTHM [BM79]. Coded in Fortran, the algorithm has a rather difficult soundness proof that demands the use of five lemmas to check the 61 verification conditions issued by a Fortran verification condition generator. Besides NQTHM, several interactive theorem provers also succeeded to prove it, for example PVS and Nuprl [How93] and STeP [Bjø98].

The idea of the algorithm is to pair off the values and to erase pairs of different values such that the returned value at the end of the erasing process is the potential majority value. MJRTY can be easily converted from an imperative program to a recursive function \( m(p, i) \) that returns a pair \((mcv, mlv)\), where \( mcv \) is the majority candidate at a certain moment and \( mlv \) is its lead over the other candidates knowing that the \( i \) votes are stored in a poll \( p \) given as input (see Algorithm 1).

---

**Algorithm 1** \( m(p, i) \): the MJRTY algorithm

**Require:** a poll \( p \) of \( i \) votes

**Ensure:** the majority candidate and its lead over the other candidates

1. if \( i > 0 \) then
2. \( (mcv, mlv) \leftarrow m(p, i - 1) \)
3. if \( p[i] = mcv \) then
4. return \((mcv, mlv + 1)\)
5. else if \( mlv > 0 \) then
6. return \((mcv, mlv - 1)\)
7. else
8. return \((p[i], 1)\)
9. end if
10. else
11. \((No, 1)\)
12. end if

The SPIKE specification of MJRTY is in Figure 2. It consists of four main parts. The first part is devoted to the specification of the sorts: \( \text{nat} \) for the naturals, \( \text{bool} \) for the booleans, \( \text{cand} \) for the candidates, and \( \text{list} \) for the lists of candidates. The prover is instructed to apply the cooperation schema by the instruction \textit{use: nat\(\text{s}\);}.

The second part of the specification contains the declaration of function symbols. First we declare the constructor symbols \( 0 \) and \( s \) for the naturals, next \textit{True} and \textit{False} for the booleans, \( \text{Nil} \) and \textit{Cons} for the lists of candidates and, finally, \textit{Cd} and \textit{No} for the candidates. \textit{No} is a pseudo candidate returned by the algorithm to indicate that there is no majority candidate already computed. Then, we declare the remaining function symbols. The function \( m \) is divided in two mutually recursive functions, \( mc \) and \( ml \), which compute respectively the majority candidate and its lead over the other candidates. \textit{count} \((p, i, c)\) counts the number of votes for a given candidate \( c \) from a poll \( p \) containing \( i \) votes. The other defined functions are \textit{access} \((p, n)\) which returns the \( n \)-th element of the list \( p \) and the 4-argument conditional function \textit{if}. The third part of the specification consists of the axiomatic definitions for the defined function symbols. The well-founded ordering over the terms \( < \) is a recursive path-ordering [Der82] built on the precedence over the function symbols presented in the last part.
Figure 2: The specification of MJRTY in SPIKE
The main conjecture states that \( mc(p, i) \) always returns the majority candidate whenever such a candidate exists in the poll \( p \) containing \( i \) votes:

\[
\forall p \text{ list}, V_i : \text{nat} \forall c : \text{cand} (i < 2 \times \text{count}(p, i, c) \Rightarrow c = mc(p, i)) \tag{1}
\]

An important lemma, provided by N. Shankar (according to [How93]), simplifies in a major way the proof of (1):

\[
2 \times (i_f(c_1, mc(p_1, i_1), 0, ml(p_1, i_1)) + \text{count}(p_1, i_1, c_1)) < s(i_1 + ml(p_1, i_1)) \tag{2}
\]

A detailed account of the proof of the lemma by SPIKE can be found in [Str98, Str00a]. Here we focus on steps in which the reasoning specialist plays a key role.

SPIKE starts by applying a case analysis on the literal if \((c_1 \neq mc(p_1, i_1), 0, ml(p_1, i_1))\). According to the definition of \( if \), we consider the two cases, namely \( c_1 \neq mc(p_1, i_1) \) and \( c_1 = mc(p_1, i_1) \). After rewriting \( if(c_1, mc(p_1, i_1), 0, ml(p_1, i_1)) \) with the corresponding \( if\)-axiom, we get the new conjectures:

\[
c_1 \neq mc(p_1, i_1) \Rightarrow 2 \times (ml(p_1, i_1) + \text{count}(p_1, i_1, c_1)) < s(i_1 + ml(p_1, i_1)) \tag{3}
\]

\[
c_1 = mc(p_1, i_1) \Rightarrow 2 \times (0 + \text{count}(p_1, i_1, c_1)) < s(i_1 + ml(p_1, i_1)) \tag{3}
\]

Here we focus on the proof of (3). Application of \textsc{Generate} to (3) moves it to the set of induction hypotheses and leads to the following sub-goal:

\[
No \neq mc(p_1, i_2), \text{access}(p_1, i_2) \neq mc(p_1, i_2), 0 < ml(p_1, i_2), No = \text{access}(p_1, i_2) \Rightarrow \tag{4}
\]

\[
2 \times ((ml(p_1, i_2) - 1) + s(\text{count}(p_1, i_2, No))) < s(i_2 + (ml(p_1, i_2) - 1))
\]

The application of \textsc{Constraint Contextual Rewriting} invokes the reasoning specialist (via \textsc{Entailment Check}) whose constraint store gets initialized to:

\[
\langle A : \{ No \neq mc(p_1, i_2), \text{access}(p_1, i_2) \neq mc(p_1, i_2), 0 < ml(p_1, i_2), \\
No = \text{access}(p_1, i_2), \\
2 \times ((ml(p_1, i_2) - 1) + s(\text{count}(p_1, i_2, No))) \geq s(i_2 + (ml(p_1, i_2) - 1)) \} \rangle
\]

The application of \textsc{2g} moves \( No \neq mc(p_1, i_2), \text{access}(p_1, i_2) \neq mc(p_1, i_2), No = \text{access}(p_1, i_2) \) to the \( G \)-field and application of \textsc{2l} linearizes \( 0 < ml(p_1, i_2), 2 \times ((ml(p_1, i_2) - 1) + s(\text{count}(p_1, i_2, No))) \geq s(i_2 + (ml(p_1, i_2) - 1)) \) and then adds the results to the \( P \)-field. This results in the constraint store:

\[
\langle G : \{ No \neq mc(p_1, i_2), \text{access}(p_1, i_2) \neq mc(p_1, i_2), No = \text{access}(p_1, i_2) \} \rangle \\
P : (\{1 + -1 \times ml(p_1, i_2) \leq 0, -1 \times ml(p_1, i_2) \leq 0, -1 \times count(p_1, i_2, No) \leq 0, \\
-1 \times i_2 \leq 0, 1 + -2 \times count(p_1, i_2, No) + -1 \times ml(p_1, i_2) + 1 \times i_2 \leq 0 \} \rightarrow 0) \rangle
\]

Since contradiction has not been derived, the \textsc{Augment} rule is then applied using the induction hypothesis (3) to promote further inference with the inequality \( 1 + -2 \times count(p_1, i_2, No) + -1 \times ml(p_1, i_2) + 1 \times i_2 \leq 0 \). SPIKE instantiates (3) with the substitution \( \{ c_1 \mapsto No, i_1 \mapsto i_2 \} \) thereby obtaining the following instance:

\[
No \neq mc(p_1, i_2) \Rightarrow 2 \times (ml(p_1, i_2) + \text{count}(p_1, i_2, No)) < s(i_2 + ml(p_1, i_2)) \tag{5}
\]

(The condition \( No \neq mc(p_1, i_2) \) is readily proved by the prover since it is a trivial consequence of the constraint store.) The extension of the constraint store with the linearization of \( 2 \times (ml(p_1, i_2) + \text{count}(p_1, i_2, No)) < s(i_2 + ml(p_1, i_2)) \) yields a constraint store containing the impossible inequality \( 1 \leq 0 \) in the \( P \)-field. The unsatisfiability of the constraint store is then easily detected by \textsc{ca-unsat}.

\[5\] To simplify the notation we represent only the non-empty fields of the structure and we tag the non empty field with the corresponding name.
8 Conclusions and Future Work

We have presented a general scheme for the integration of decision procedures with an implicit induction prover. The integration scheme is effective since when applied with Spike and decision procedures for equality and arithmetics it has given positive results on several non-trivial problems. Moreover, the soundness of our integration has been formally derived; this task is not obvious since we allow some interleaving between induction hypothesis application and decision-procedure application.

We plan to apply the integration scheme to new decision procedures for new decidable theories such as those for lists and arrays [ARR01]. At the same time, we will apply the theorem-prover to the verification of protocols like ABR [RSK00]. We would like also to exploit better for efficiency built-in AC operators by using AC-matching as in [BBR96].

References


The AutoBayes Program Synthesis System
— System Description —

Bernd Fischer, Thomas Pressburger, Grigore Rosu, and Johann Schumann

RIACS / Code IC, NASA Ames Research Center, M/S 269-2
Moffett Field, CA 94035 USA;
{fisch, ttp, grossu, schumann}@ptolemy.arc.nasa.gov

1 Introduction

AutoBayes is a fully automatic program synthesis system for the statistical data analysis domain. Its input is a concise description of a data analysis problem in the form of a statistical model; its output is optimized and fully documented C/C++ code which can be linked dynamically into the Matlab and Octave environments. AutoBayes synthesizes code by a schema-guided deductive process. Schemas (i.e., code templates with associated semantic constraints) are applied to the original problem and recursively to emerging subproblems. AutoBayes complements this approach by symbolic computation to derive closed-form solutions whenever possible. In this paper, we concentrate on the interaction between the symbolic computations and the deductive synthesis process; a detailed description of AutoBayes can be found in [FSP00, FS01].

A statistical model specifies for each problem variable (i.e., data or parameter) its properties and dependencies in the form of a probability distribution. A typical data analysis task is to estimate the best possible parameter values from the given observations or measurements. The following example models normal-distributed data but takes prior information (e.g., from previous experiments) on the data’s mean value and variance into account.

```plaintext
1 model normal as 'Normal model with conjugate priors'.
2 const double kappa_0, mu_0.
3 where 0 < kappa_0.
4 double mu ~ gauss(mu_0, sqrt(sigma_sq/kappa_0)).
5 const double sigma_0_sq, delta_0.
6 where 0 < sigma_0_sq and 0 < delta_0.
7 double sigma_sq ~ invgamma(delta_0/2+1, sigma_0_sq*(delta_0/2)).
8 const nat n_points.
9 where 0 < n_points.
10 data double x(0..n_points-1) ~ gauss(mu, sqrt(sigma_sq)).
11 max pr({x, mu, sigma_sq}) for {mu, sigma_sq}.
```

Here, lines 8–10 describe the data properties: x is a vector of n_points real-valued observations that are independently drawn from a normal or Gaussian distribution with unknown mean mu and variance sqrt(sigma_sq). Lines 2–4 specify the prior information on mu, which is itself drawn from a normal distribution. This prior summarizes a number of previous experiments, where mu turned out to be mu_0 on average. Similarly, lines 5–7 specify the prior on sigma_sq. Lines 2, 5, and 8 declare constants to represent the model parameters; lines 3, 6, and 9 state constraints on their allowed values.
Finally, line 11 comprises the proper analysis task: given the data \( x \), find the values for \( \mu \) and \( \text{sigma}_\text{sq} \) which maximize the joint probability \( p(x; \mu, \text{sigma}_\text{sq}) \)—in other words, find the values for \( \mu \) and \( \text{sigma}_\text{sq} \) which explain the observed data \( x \) in the statistically best possible way.

**Graphical models** [Bun94] are a uniform framework in which many typical data analysis problems, e.g., data compression [Fre98] or image restoration [Kok98], can be formulated as similar parameter learning problems. AUTOBAYES uses graphical models to represent models internally and to guide the decomposition of the statistical learning task into simpler, independent subtasks.

# System Architecture

Program generation proceeds in a number of distinct stages that are reflected in AUTOBAYES’s system architecture. In a preprocessing step, the given specification is parsed and converted into the internal graphical model form. The synthesis kernel then analyzes the model and tries to solve the given optimization task. It instantiates appropriate algorithm schemas which are given in a library and produces a procedural program in AUTOBAYES’s intermediate language. This code is optimized and finally converted into the language of the target system. The synthesized code is fully documented; assumptions and proof obligations which have not been discharged during synthesis are laid out clearly in the documentation or are converted into runtime assertions.

The entire system is implemented in SWI-Prolog [Wie98] and comprises about 24,000 lines of documented code. SWI-Prolog proved to be a very stable and efficient development platform with reasonably good debugging facilities.

**Synthesis Kernel.** Synthesis is performed by exhaustive, layered application of schemas. A schema consists of a program fragment with open slots and a set of applicability conditions. The slots are filled in with code pieces by the synthesis kernel. The conditions constrain how the slots can be filled; they must be discharged (i.e., proven to hold in the given model) before the schema can be applied. Conditions can also be described by specific network patterns; checking then proceeds efficiently by pattern matching. This allows the network structure to guide the application of the schemas and thus to prevent combinatorial explosion of the search space, even if a large number of schemas are applicable.

AUTOBAYES currently comprises four different layers of schemas; schemas can easily be added without restructuring the system. Network decomposition schemas try to break down the network into independent subnets, based on independence theorems for graphical models. The emerging subnets are fed back into the synthesis process and the resulting programs are composed to achieve a program for the original problem. AUTOBAYES is thus able to automatically synthesize large programs by composition of different schemas. Formula and vector decomposition schemas work on products of conditional probability distributions. The application of these schemas is also guided by the network structure but they require more substantial symbolic computations. The skeleton of the synthesized code is generated by the application of statistical algorithm schemas. AUTOBAYES currently implements two such schemas, the EM-algorithm and k-Means (i.e., nearest neighbor clustering). After this last network-oriented layer, the statistical problem has been transformed into an ordinary optimization problem. If AUTOBAYES cannot find a symbolic solution for this problem, it applies standard numeric optimization methods. AUTOBAYES currently provides schemas for the Newton-Raphson and Nelder-Mead simplex algorithms. These schemas are instantiated with
the function to be optimized. In contrast to using a library function, this open approach allows further symbolic simplifications and optimizations, as well as problem specific documentation.

**Symbolic Subsystem.** The main task of this subsystem is to find symbolic solutions to optimization problems. This daunting task, however, is simplified substantially by the relatively uniform structure of the optimization problems which allows implementing powerful heuristics.

At the core of the symbolic subsystem is a small but reasonably efficient A.C.-rewrite engine implemented in Prolog. Since a rewrite system for this engine is implemented naturally as a Prolog-predicate, conditional rewriting comes “for free.” Moreover, the rule clauses can access explicit assumptions; hence, AUTOBAYES allows conditional rules as for example \( x/x \rightarrow \gamma_{\mu, x \neq 0} 1 \) where \( \gamma_{\mu, x \neq 0} \) means “rewrites to, provided \( x \neq 0 \) can be proven from the current assumptions.” The assumptions are managed almost transparently by the rewrite engine; the rewrite system only needs to contain the non-congruent propagation rules which modify the assumptions under which subterms are rewritten, e.g., \( \text{if } p \text{ then } s \text{ else } t \text{ fi } \rightarrow \gamma_{\mu, A} \) if \( p \downarrow_{\mu, A} \) then \( s \downarrow_{\mu, A \land p} \text{ else } t \downarrow_{\mu, A \land \neg p} \text{ fi } \) where \( t \downarrow_{\mu, A} \) is the normal form of \( t \) under the assumptions \( A \).

Expression simplification and symbolic differentiation are implemented on top of the rewrite engine. The basic rules are straightforward; however, vectors and matrices introduce the usual aliasing problems and require careful formalizations. For example, as the index values \( i \) and \( j \) are usually unknown at synthesis time, the partial derivative \( \partial x_i / \partial x_j \) can only be rewritten into \( \text{if } i = j \text{ then } 1 \text{ else } 0 \text{ fi } \). More advanced rules, however, require explicit meta-programming, especially when bound variables are involved.

Abstract interpretation is used as an efficient mechanism to evaluate range constraints such as \( x > 0 \) or \( x \neq 0 \) which occur in the conditions of many rewrite rules. AUTOBAYES implements as a rewrite system a domain-specific refinement of the standard sign abstraction where numbers are not only abstracted into \( \text{pos} \) and \( \text{neg} \) but also into \( \text{small} \) (i.e., \( |x| < 1 \)) and \( \text{large} \).

It then turns out that a relatively simple solver built on top of this core system is already sufficient. AUTOBAYES thus essentially relies on a low-order polynomial (i.e., linear, quadratic, and simple cubic) symbolic solver. However, it also shifts and normalizes exponents, recognizes multiple roots and bi-quadratic forms, and tries to find polynomial factors. It also handles expressions in \( x \) and \( (1 - x) \) which are common in Bernoulli models.

## 3 Experimental Results

We have applied AUTOBAYES to a number of advanced textbook examples, machine learning benchmarks, and NASA applications. Table 1 summarizes the results. \( cfs \) indicates whether a closed-form solution exists and, if so, whether it was found by AUTOBAYES. The remaining columns give the size of the specification and the respective number of lines of generated Octave/C++ code, as well as synthesis and compilation (\( g++ \) -O2) times on a Sun Ultra 60.

The examples \( N_1 \) to \( N_4 \) describe different estimation problems for normal distributions where \( N_3 \) is the example in Section 1. \( N_1 \) and \( N_2 \) are simpler versions where prior information is specified only for the mean or the variance of the data. These are advanced textbook examples [GC’95] and AUTOBAYES finds exactly the closed form textbook solutions. \( N_4 \) slightly generalizes the form of the prior for \( \mu \). However, this seemingly small modification renders the symbolic problem unsolvable. AUTOBAYES
The AutoBayes Program Synthesis System — System Description —  171

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>cfs</th>
<th>lines of spec C++</th>
<th>$T_{synth}[s]$</th>
<th>$T_{compile}[s]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$N_1 \mu \sim N(\mu_0, \sigma_0^2) \cdot \sigma^2$</td>
<td>Y</td>
<td>8</td>
<td>99</td>
<td>1.5 + 7.1</td>
</tr>
<tr>
<td>2</td>
<td>$N_2 \mu, \sigma^2 \sim \Gamma^{-1}(\delta_0/2 + 1, \sigma_0^2)$</td>
<td>Y</td>
<td>9</td>
<td>99</td>
<td>2.0 + 8.8</td>
</tr>
<tr>
<td>3</td>
<td>$N_3 \mu \sim N(\mu_0, \sigma_0^2 \cdot \kappa_0)$</td>
<td>Y</td>
<td>12</td>
<td>126</td>
<td>8.9 + 7.7</td>
</tr>
<tr>
<td>4</td>
<td>$N_4 \mu \sim N(\mu_0, \sigma_0^2) \cdot \sigma^2 \sim \Gamma^{-1}(\delta_0/2 + 1, \sigma_0^2 \cdot \delta_0/2)$</td>
<td>No</td>
<td>12</td>
<td>478</td>
<td>14.6 + 20.0</td>
</tr>
<tr>
<td>5</td>
<td>$M_1$ 1D Gaussian mixture</td>
<td>No</td>
<td>16</td>
<td>389</td>
<td>11.7 + 12.4</td>
</tr>
<tr>
<td>6</td>
<td>$M_2$ 2D Gaussian mixture (x, y uncorrelated)</td>
<td>No</td>
<td>22</td>
<td>536</td>
<td>19.6 + 19.7</td>
</tr>
<tr>
<td>7</td>
<td>$M_3$ 1D Gaussian mixture (multi-dim. classes)</td>
<td>No</td>
<td>24</td>
<td>519</td>
<td>18.1 + 16.7</td>
</tr>
<tr>
<td>8</td>
<td>$M_4$ exponential mixture (simple failure analysis)</td>
<td>No</td>
<td>15</td>
<td>321</td>
<td>6.4 + 10.0</td>
</tr>
<tr>
<td>9</td>
<td>$M_5$ disjoint mixture (binomial + Poisson)</td>
<td>No</td>
<td>21</td>
<td>425</td>
<td>19.5 + 11.9</td>
</tr>
<tr>
<td>10</td>
<td>SD step detection</td>
<td>No</td>
<td>14</td>
<td>1206</td>
<td>78.0 + 49.4</td>
</tr>
<tr>
<td>11</td>
<td>AB Abalone classifier</td>
<td>No</td>
<td>58</td>
<td>1310</td>
<td>63.5 + 139.1</td>
</tr>
<tr>
<td>12</td>
<td>GR $\gamma$-ray burst analysis</td>
<td>No</td>
<td>12</td>
<td>475</td>
<td>3.9 + 9.5</td>
</tr>
</tbody>
</table>

Table 1: List of examples

thus generates executable code by instantiation of an iterative numeric solver. This example shows that a purely symbolic system is not sufficient in practice.

The remaining examples all require the application of iterative algorithms. The examples $M_1$ to $M_5$ are all solved via the EM algorithm schema but each example induces a different symbolic maximization problem. However, after symbolic differentiation, these subproblems are reduced to essentially linear or quadratic equations which are easily solved by AUTOBAYES. $SD$ is a simple time series model to detect a change of means in a Gaussian process; $AB$ is a classifier for abalone mussels (http://www.ics.uci.edu/~mlearn/MLRepository.html). Finally, $GR$ is a model to detect $\gamma$-rays bursts from the BATSE radio source (http://cossr.gsfc.nasa.gov/batse).

4 Conclusions

The tight combination of schema-guided synthesis, deduction, and symbolic computation in AUTOBAYES is essential to generate efficient code. Symbolic computation is used for simplification and for finding symbolic solutions if they exist. However, we can only synthesize a correct program from a specification when we can rely on the soundness of the symbolic machinery. This in particular means that all transformations have to be performed with respect to the proper assumptions, like an expression being non-zero. Transformations can also give rise to new proof obligations, e.g., showing that a possible solution is the minimum and not just a saddle point. AUTOBAYES keeps track of all assumptions and either discharges them during synthesis or generates assertions to be checked during runtime. The importance for symbolic calculation under assumptions and the unsoundness of a commercial symbolic algebra system like Mathematica led us to develop our own symbolic subsystem on top of Prolog.

Although we have been able to synthesize code for various non-trivial textbook examples, AUTOBAYES's code generating capabilities for a variety of statistical models need to be extended substantially. Besides adding further algorithm schemas for statistical computations and for general numerical optimizations, improvement of the symbolic subsystem is of major importance. The power and generality of the equation
solver will need to be enhanced. Furthermore, for marginalization in statistical models, symbolic handling of (relatively) simple integrals is important. Each enhancement in the symbolic subsystem will lead to improvement of the synthesized code as more subtasks can be solved in closed form rather than being approximated by (slower) numerical algorithms. In all cases, AutoBayes ensures correctness of the synthesized code with respect to the specification by generating the appropriate runtime assertions and documentation.

References

The ActiveMath Learning Environment
System Description

Paul Libbrecht, Erica Melis, Carsten Ulrich
paul.melis.ulrich@ags.uni-sb.de
Saarland University, Germany

Abstract ActiveMath is a web-based learning environment that dynamically generates interactive mathematical courses adapted to the student's goals, preferences, capabilities, and knowledge. It integrates several mathematical service systems. The course content is represented in OMDoc, an extension of OpenMath. For each user, the appropriate content is retrieved from a knowledge base and the actual course is generated individually according to pedagogical rules. The course is presented to the user via a standard web-browser. During a course the learner can interactively practice problem solving by using mathematical services such as computer algebra system or a proof planner. The article provides a brief account of the current state of ActiveMath.

1 Introduction

Web-based learning systems are becoming more predominant and increasingly incorporate intelligent features. In Saarbrücken we are developing the web-based, user-adaptive, interactive learning environment ActiveMath. In a nutshell, its major features are user-adapted content, sequencing, and presentation, user-adapted suggestions for learning, support of active and explorative learning by mathematical services, use of proof planning, support of teachers by information about their students, and the reusability of the encoded content via the semantics of the knowledge representation.

We believe that the use of such a system in the maths curriculum is advantageous as it supports active learning of the student. During the last decades, the mathematics pedagogy community recognized that students learn mathematics more effectively, if the traditional rote learning of formulas and procedures is supplemented with the possibility to explore a broad range of problems and problem situations [11]. In particular, the international comparative study of mathematics teaching, TIMSS [1], has shown (1) that teaching with an orientation towards active problem solving yields better learning results in the sense that the acquired knowledge is more readily available and applicable especially in new contexts and (2) that a reflection about the problem solving activities and methods yields a deeper understanding and better performance.

We also believe that teaching mathematical methods and know-how and know-when has to be introduced into mathematics teaching, apart from the traditional axioms, theorems, and procedure teaching. The knowledge-acquisition work done for proof planning provides a first material that has to be further expanded. First experiments [9] suggest that instruction materials based on descriptions of mathematics methods

\footnote{A demonstration of ActiveMath is available at http://www.activemath.org}
yield a better subsequent problem solving performance than traditional (textbook like) instruction material.

This article provides a brief account of the current state of ActiveMath. In particular, we focus on the architecture, the knowledge representation, the presentation planning, and the user model. For a more detailed description see [8].

2 Architecture

Figure 1 depicts the architecture of ActiveMath, i.e., its components and the communications between them (indicated by arrows). It shows the client-server web-architecture with a browser at the client side. Currently, ActiveMath integrates the following components: a session manager, the knowledge base MBase [4], a presentation planner, a user model, a pedagogical module, and mathematical services such as the proof planner of Omega [16] and the Computer Algebra System (CAS) MAPLE (for a detailed description of the integration of mathematical services see [7]). The components can communicate over the Internet via a standardized XML-RPC protocol. For instance, requests of the user and (in the other direction) HTML-pages are communicated via a web-server to the session manager. The session manager stores the generated courses and translates URL requests into actions (e.g., the request for a new course about the topic group) that are passed to the responsible component. The presentation planner generates the learning documents adapted to the user's goals, preferences, and knowledge by requesting and processing information from MBase, from the user model, and from the pedagogical module. Information about the user's actions, such as the history of her actions and the time intervals of her reading a concept or the success of solved
problems, is passed from the session manager to the user model (as well as from the
proof planner and CAS to the user model), where it is used for updating.

3 Knowledge Representation

Our knowledge representation OMdoc [6], is an extension of the OpenMath [3] standard. OMdoc encodes the semantic of mathematical objects as well as meta-data and mathematical facts such as theorems, definitions, proof methods, and proofs and includes natural language formulations as well as formal (OpenMath) objects. OMdoc uses a semantic XML-based representation of mathematical knowledge that provides an ontology for the content of the course which is indispensable for a reuse of teaching and learning material and for a combination of material from different sources.

In addition to the actual conceptual content, our knowledge representation contains meta-data for structures, dependencies, and pedagogical information which can be used for the dynamic generation of interactive documents.

4 Presentation Planning

The central component of ActiveMath is the presentation planner. It generates a personalized course in a three-stage process:
(1) First, the content is retrieved from the knowledge base MBASE. Starting from the goal concepts chosen by the user, all concepts they depend upon and corresponding additional information (e.g., elaborations, examples for a concept) are collected recursively. This grabbing process uses the dependencies metadata information contained in the OMdoc representation. The result of this retrieval is a collection of all concepts plus additional information about them that need to be known by the learner in order to understand the goal concepts.
(2) Then pedagogical knowledge is applied. According to the information in the user model and the pedagogical module the collection of content items is processed and transformed into a personalized instructional graph of learning materials. This process is detailed below.
(3) Finally, the instructional graph is linearized.

The result of the presentation planning is a linearized instructional graph whose nodes are OMdoc items. Filters (see Figure 1) will transform this collection into HTML pages using XSL-transformations.

The goal of the application of pedagogical knowledge is to select from and transform the collection of items that was gathered in the first stage of presentation planning into a selection of learning material. ActiveMath employs pedagogical information represented in pedagogical rules. It evaluates the rules with the expert system shell JESS [5]. The rules consist of a condition and an action part. The condition part of a rule specifies the conditions that have to be fulfilled for the rule to be applied, the action part specifies the actions to be taken when the rule is applied.

The presentation planner employs the pedagogical rules to decide:
(1) which information should be presented on a page;
(2) in which order this information should appear on a single page;
(3) how many exercises and examples should be presented and how difficult they should be;
(4) whether or not to include exercises and examples that make use of a particular service system. Since exercises and examples employing service systems require a certain minimal familiarity with the systems, ACTIVEMATH presents those exercises only if the capability is confirmed.

(5) whether to restrict the available functionalities of a service system. For instance, a student learning about mathematical integration and derivation should not use a CAS to solve his exercises completely, whereas using the CAS as a calculator for auxiliary calculation is acceptable.

5 User Modeling

The user model consists of two subcomponents: the history that stores the data about user’s actions (e.g., the reading of a concept at a certain time), and the profile that stores the user’s preferences, goals and knowledge mastery data. When a learner registers, she can assert her knowledge mastery values which will be constantly updated according to her actions. The user model is inspectable and modifiable.

The knowledge mastery assessment is represented by values for a subset of the competence features in Bloom’s taxonomy [2], namely Knowledge, Comprehension, and Application.

Depending on the type of the user’s interaction, different updates of the values are realized. After reading a concept, mainly the Knowledge-value of it is updated. After reading an example for a concept the Comprehension-value of that concept is mainly updated. After acting on an exercise, the success or failure rate mainly updates the Application-value of that concept in a way depending on the abstractness and difficulty levels of the exercise. A dependency between those values will be introduced later.

The user model also stores a justification for each value, that is, whether it is a direct user input for the user model, indirectly inferred from the history data or other information, or an output of an exercise evaluator.

6 Conclusion and Further Work

This paper gives a short overview on the web-based learning environment ACTIVEMATH that generates mathematical courses adapted to the learner and integrates several mathematical service systems.

We plan to integrate other mathematical systems, such as the CAS MuPad. An interesting question regarding mathematical systems is how to interpret the learners actions and how to calculate the appropriate feedback to the user model, preferably in an abstract way in order to avoid having to build a proper evaluator for each mathematical system.

Other questions concern metadata and presentation planning. We are redesigning the current prototypical version of the proof planner and have to decide what kind of metadata is needed to generate useful learning material.

Last, but not least evaluation is an important issue. We are conducting studies on the design of the user interface and prepare field studies in schools and at university to test whether our system does what it is supposed to do, support learning.
References

System Description: Interface between Theorema and External Automated Deduction Systems*

Temur Kutsia and Koji Nakagawa

Research Institute for Symbolic Computation
Johannes Kepler University Linz
A-4040, Linz, Austria
{kutsia,nakagawa}@risc.uni-linz.ac.at

Software Competence Center Hagenberg
Hauptstrasse, 99
A-4232, Hagenberg, Austria
{teimuraz.kutsia,koji.nakagawa}@scch.at

Abstract The interface between the Theorema system and external automated deduction systems is described. It provides a tool to access external provers within a Theorema session in the same way as "internal" Theorema provers. Currently 11 external systems are supported. The design of the interface allows combining external systems with each other as well as with "internal" Theorema provers.

1 Introduction

The Theorema system [7] is an integrated environment for proving, solving and computing built on the top of mathematical software system Mathematica [26]. It was designed by Bruno Buchberger [5, 6] and provides a front end for composing formal mathematical text consisting of a hierarchy of axioms, definitions, propositions, algorithms etc. in a common logic frame with user-extensible syntax, a library of both well-established and new provers, solvers and simplifiers for proving, solving and simplifying mathematical formulae.

The interface described in this paper implements a link providing Theorema users with a tool for using automated deduction systems ("external" systems) within Theorema session in the same way as "internal" Theorema provers. Currently, the following 11 external systems are supported: provers - Bliksem [19], EQP [17], E [21], Gandalf [23], Otter [16], Scott [12], Sether [15], Spass [24], Vampire [20], Waldmeister [4] and a finite model and counter-example searcher Mace [18].

The interface consists of two types of links - direct and indirect - from Theorema to an external system. The indirect link is established with Sether, Scott and Waldmeister - using first an intermediate translation into TPTP [22] format, and then the TPTP2X converter which converts TPTP format files into a format of the specified external system. All the other provers are linked directly to Theorema, without any intermediate routine, translating input from Theorema syntax directly into the syntax of an external system. It is relatively easy to add an indirect link to a new prover, but it has also disadvantages compared with the direct link - the user has less control over the system and more intermediate routines are needed.

* Supported by the Austrian Science Foundation (FWF) under Project SFB F1302 and by Software Competence Center Hagenberg (Austria) under MathSoft project.
All the external provers can be used as black box provers within Theorema session, without translating their output into Theorema syntax. The link to Otter has an additional feature - back translator, which transforms an Otter proof into Theorema format. The design of the interface makes it possible to combine various external systems with each other or with Theorema provers.

2 Direct Link

The interface implements two types of direct links from Theorema to external systems: black box style and white box style links. A black box style direct link consists of two parts: the translator component and the linking component. The components and the sequence of operations are illustrated in Figure 1. A white box style direct link consists of the translator, the linking component and the back translator. The architecture is shown in Figure 2.

The black box style link works as follows: first, the translator gets the Theorema goal, knowledge base and options and translates them into the prover format thus preparing the input for the prover call. The linking component gets the translated string and options, writes the string in a temporary file and calls the prover with the options. Finally, the prover output is passed back to Theorema. The user has a full control over the external system using prover options.

The sequence of operations performed by the white box style direct link is the same as those of the black box style link until the external prover output is passed back. Here the back translation component gets the prover output, transforms it into Theorema syntax and places into the Theorema proof object. Finally, the Theorema style proof is shown in the proof notebook. Note that if the prover failed to prove the goal, the back translation component does not affect the output and the link behaves like a black box style link. Currently Theorema has black box style links with Otter, Spas, EQLP, Gandalf, Bliksem, Vampire, E and Mace and the white box style link with Otter only. Thus, Otter has both types of direct links to Theorema, with the black box style link as the default option. The notebooks on the figures above show the black box and the white box outputs of the Otter call to prove that every Robbins algebra satisfying idempotence of addition is a Boolean algebra ([25]). In both cases, the output in Otter syntax can be shown in a new notebook which pops up if one clicks on the corresponding hyperlink in the proof notebook. In the back translated output each inference rule used in the proof has a hyperlink to the definition of it.

Figure 1: Black box style direct link

Figure 2: White box style direct link
3 Indirect Link

All indirect links from Theorema to external systems are of the black box style. An indirect link consists of three parts: the translator into TPTP format, the linking component to TPTP2X and the linking component to the external prover. We have chosen the TPTP format for the intermediate translation because of the tptp2X converter, which can convert a TPTP format file into a format of many automated provers. The indirect link works as follows: the Theorema Prove call invokes the translator into TPTP format which translates the goal and knowledge base into TPTP format and puts the result into a temporary file. Next, the linking component to TPTP2X calls the script tptp2X provided with the TPTP library on the file with the command line options relevant to the prover. The obtained result is passed to the linking component to the prover together with the options. The linking component calls the prover, gets the result back and locates it into the Theorema proof object. It is relatively easy to add an indirect link to a new prover, but it has also disadvantages comparing with the direct link - the user has less control over the system and an intermediate translation/linkage is needed. In future, we intend to replace indirect links with direct ones. The indirect link is established with Setheo, Scott and Waldmeister.

4 Combination of Systems

The design of the interface allows combining various external systems with each other or with “internal” Theorema provers in the similar way as the “internal” provers can be combined with each other. To demonstrate this capability we combined Otter and Mace into the Theorema user prover OtterMace (not the prover from Argonne), which first runs Otter on a problem, and if it fails to prove the goal, Mace tries to find a countermodel. Another example of such a combination is the Theorema experimental user prover PLOtter, which combines Otter and the predicate logic prover of Theorema.
5 Related Work and Conclusion

The problem of integrating software/reasoning systems has been studied intensively in the recent years (e.g. [2], [3], [10], [11], [13], [14]). There are several platforms which try to provide a general solution to the problem ([11], [8], [9]). The interface described in this paper is not an attempt to provide such a general solution. We tried to make an interface which is simple, easy to implement, maintain and use and which links reasoning systems directly, without intermediate routines (we still have indirect links for experimental reasons but we intend to replace them with direct ones). The interface allows using external systems in a Theorema session as "internal" Theorema provers.

References