The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

IJCAR'06 Workshop

UITP'06: User-Interfaces for Theorem Provers

August 21st, 2006

Proceedings

Editors:

S. Autexier and C. Benzmüller

Table of Contents

Preface

This volume contains the papers of the 7th Workshop on *User Interfaces for Theorem Provers (UITP 2006)*, which was held on 21st of August, 2006, in Seattle, Washington, USA. UITP 2006 was affiliated with the *International Joint Conference on Automated Reasoning (IJCAR 2006)* and organized as a part of *The 2006 Federated Logic Conference (FLoC 2006)* which took place in Seattle from 10th to 22nd of August, 2006.

The *User Interfaces for Theorem Provers* workshop series brings together researchers interested in designing, developing and evaluating interfaces for interactive proof systems, such as theorem provers, formal methods tools, and other tools manipulating and presenting mathematical formulas. UITP provides a forum for all those interested in improving human interaction with and usability of proof systems.

The first workshop in the UITP series was held in 1995 in Glasgow (organized by Phil Gray, Tom Melham, Muffy Thomas and Stuart Aitken). Further meetings took place in 1996 in York (organized by Nicholas Merriam, Michael Harrison and Andy Dearden), 1997 in Antibes (organized by Yves Bertot) and 1998 in Eindhoven (organized by Roland Backhouse). There followed a break until 2003 when the workshop was revived by David Aspinall and Christoph Lüth and organized as part of the *Theorem Proving in Higher Order Logics (TPHOLs 2003)* conference in Rome. UITP 2005 (again organized by David Aspinall and Christoph Lüth) was held as a satellite workshop of the *European Joint Conferences on Theory and Practice of Software (ETAPS 2005)* in Edinburgh. The present organizers were asked to orga-

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

nize another UITP workshop in 2006 and we decided to do this in affiliation with IJCAR at this years FLoC.

The high quality papers published in this volume (and the previous ENTCS UITP volumes) document the liveliness and innovative strength of the UITP community. Despite the recently increasing interest in better interfaces for theorem provers and mathematical support tools, however, we are convinced that the UITP initiative requires and deserves a regular and sustainable fostering and also better visibility in and communication with the wider *Artificial Intelligence* community.

We would like to thank several people who helped us in the organization of this workshop. First of all, many thanks to all program committee members

David Aspinall	Ewen Denney	Florina Piroi
Yves Bertot	Christoph Lüth	Aarne Ranta
Paul Cairns	Michael Norrish	Makarius Wenzel

and all additional reviewers

Loïc Pottier	Laurence Rideau	Hua Yang

for their support and productive collaboration. Many thanks to the organizers of FLoC'06 and IJCAR'06, in particular Tom Ball, for setting up the FLoC workshop environment, and Andrei Voronkov, for his EasyChair conference tool that simplified our organizational work a lot. Last but not least, many thanks to all authors who submitted papers and to all active participants at the workshop.

Serge Autexier and Christoph Benzmüller
Edinburgh and Saarbrücken, July 2006

A Graphical User Interface for Formal Proofs in Geometry

Julien Narboux[1]

LIX, École Polytechnique
91128 Palaiseau, France

Abstract

We present in this paper the design of a graphical user interface to deal with proofs in geometry. The software developed combines three tools: a dynamic geometry software to explore, measure and invent conjectures, an automatic theorem prover to check facts and an interactive proof system (Coq) to mechanically check proofs built interactively by the user.

Keywords: geometry, theorem prover, proof assistant, interface, Coq, dynamic geometry, automated theorem proving

1 Introduction

Dynamic Geometry Software (DGS) and Computer Algebra Software (CAS) are the most widely used software for mathematics in the education. DGS allow the user to create complex geometric constructions step by step using free objects such as free points and predefined atomic constructions depending on other objects (for instance the line passing through two points, the midpoint of a segment, *etc.*). The free objects can be dragged using the mouse and the figure is updated in real time. CAS allow symbolic manipulations of mathematical expressions.

The most widely used systems are the historical ones which appeared in the 90s, namely Geometer's sketchpad [22] and Cabri Geometer [26]. But there exists a large number of free and commercial software as well [2].

The education community has studied the impact of the use of these software on the *proving* activity [41,16]. DGS are used for mainly two activities:

- to make the student create geometric constructions;

[1] Email: Julien.Narboux@inria.fr

[2] We can cite (the list is not intended to be exhaustive): CaR, Chypre Cinderella, Déclic, Defi, Dr. Geo, Euclid, Euklid DynaGeo, Eukleides, Gava, GeoExp, GeoFlash, GeoLabo, GeoLog, Geometria, Geometrix, Geometry Explorer, Geometry Tutor, GeoPlanW, GeoSpaceW, GEUP, GeoView, GEX, GRACE, KGeo, KIG, Mentoniezh, MM-Geometer, Non-Euclid, XCas, *etc.*

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- to make the student explore the figure and conjecture and check facts.

We believe that these software should also be used to help the student in the proving activity itself. Work has been performed in this direction and several DGS with proof related features have been produced. Theses systems can be sorted in in roughly two categories:

(i) the systems which permit to build proofs;

(ii) the systems which permit to check facts using an automated theorem prover.

The *Geometry Tutor* [3], *Mentoniezh* [33], *Defi* [1], *Chypre* [8], *Cabri-Euclide* [27], *Geometrix* [19] and *Baghera* [6] systems belongs to the first category. Using these systems the student can produce proofs interactively using a set of known theorems. In most of these systems the student can not invent a proof very different from what the program had pre-computed using automated theorem proving methods. As far as we know, the exception is *Cabri-Euclide* which contains a small formal system and therefore gives more liberty to the student. *Baghera* includes also e-learning features, such as task management and network communication between teachers and their students.
MMP-Geometer[18],*Geometry Expert* [17], *Geometry Explorer*[36] and *Cinderella* [24,25,34,35] belongs to the second category. *Geometry Expert* and *MMP-Geometer* are DGS which are used as a graphical interface for an implementation of the main decision procedures in geometry. *Geometry Explorer* provides a diagrammatic visualization of proofs generated automatically by a prolog implementation of Chou's full angle method [14]. *Cinderella* includes a "probabilistic theorem prover" to allow the user to check facts and allows to export the description of the figure to computer algebra software to perform algebraic proofs.

The work closest to ours is [9]. The *GeoView* software provides a visualization tool for some formal geometric statements using an off-the-shelf DGS and the PCoq user interface for Coq [10,2]. It is intended to be used with the formalization of geometry for the French curriculum by Frédérique Guilhot [20] in the Coq proof assistant [15].

We present in this paper the design of a system whose aim is to combine automatic theorem proving, interactive theorem proving using a formal proof system (the Coq proof assistant) and diagrammatic visualization. The difference between our approach and the other systems we have cited (except GeoView) is that we use of a general purpose proof assistant and combine interactive and automated theorem proving. The difference between our system and GeoView is that communication with Coq goes in the other direction.

Our approach is guided by the following motivations:

- It is very natural in geometry to illustrate a proof by a diagrammatic representation and even sometimes a diagram can be seen as a high level description of a proof [7,23,29,36,37,38]. But sometimes a diagram can be misleading. That is why the verification of the proof by a formal proof system is crucial as it provides a very high level of confidence.
- Compared to an *adhoc* proof system specialized in geometry, the use of a general purpose proof assistant such as the Coq proof assistant provides a way to combine

geometrical proofs with larger proofs. For example, it is possible to use the Coq system to prove facts about polygons by induction on the number of edges, or facts about transformations using complex numbers.

- There are facts than can not be visualized graphically and there are facts that are difficult to understand without a graphical representation. Hence, we need to combine both approaches.
- We should have both the ability to make arbitrarily complex proofs or to use a base of known lemmas, depending on the level of the user/student.

We will first give a short introduction of our prototype named GeoProof. Then we will focus on the proof related features of GeoProof: *automatic* theorem proving and interactive generation of Coq statements.

2 An overview of GeoProof

GeoProof is a free and open source Dynamic Geometry Software. It is distributed under the term of the GPL Version 2 license. It has been implemented by starting from a project called DrGeoCaml initially developed by *Nicolas François*. GeoProof is written in the *Ocaml* programming language using only portable libraries in such a way that it can be compiled for Linux, Windows and MacOSX. GeoProof permits the main geometric constructions and transformations involving points, circles and lines. The documents are saved using an open format based on the XML technology. It can export the figures using a bitmap (PNG, BMP, JPEG) or vector graphic format (SVG). The figure description can also be exported to the input language of the Eukleides software to ease the insertion of figures in a LaTeX document [3]. Figure 1 gives a quick overview of the graphical user interface of GeoProof. But its main features consist in the proof oriented functionality, which will be described in the next sections.

3 Automatic proof

We present in this section how GeoProof can communicate with automatic theorem proving tools. We have implemented automatic theorem proving in GeoProof using two different systems: the first one takes advantage of an implementation of the Gröbner basis and Wu methods [40,11] written by John Harrison [21] [4], the second one consist in exporting to our own implementation of Chou's decision procedure for affine geometry [13] in the Coq proof assistant [30].

3.1 Using embedded automatic theorem prover

The formalization used by John Harrison is based on a theory with only points as basic objects whereas GeoProof uses points, lines and circles as the basic mathematical objects. We need to translate from one language to the other one. The input of the ATP is a first order formula with the following predicates: *collinear*, *parallel*,

[3] http://www.eukleides.org/

[4] Warning this implementation was designed to accompany a textbook on automated theorem proving and is not intended to be efficient.

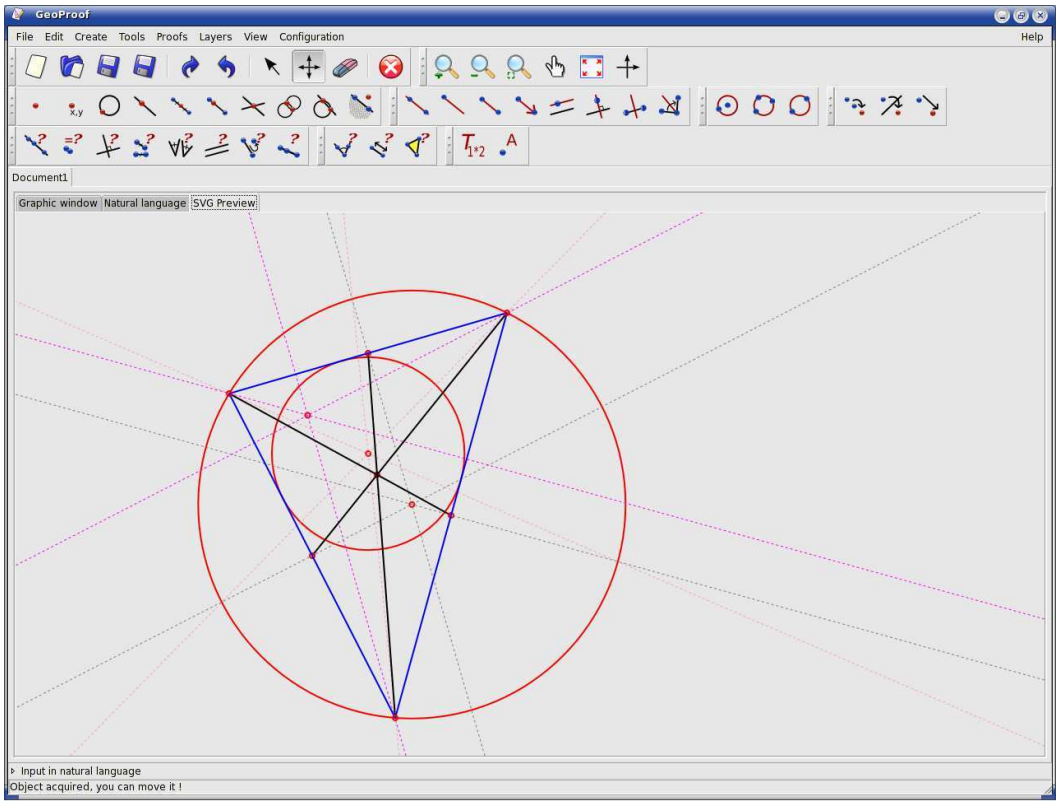


Fig. 1. A screen-shot of GeoProof, the example displayed represent the points of interest of a triangle.

perpendicular, *eq_distance* (written as $AB = CD$) and *eq_angles*. These predicates are defined using an algebraic formula using the coordinates of the points. Let x_P and y_P be the x and y coordinates of P.

$$\begin{aligned}
col(A,B,C) &\equiv (x_A - x_B)(y_B - y_C) - (x_B - x_C)(y_A - y_B) = 0\\
par(A,B,C,D) &\equiv (x_A - x_B)(y_C - y_D) - (x_C - x_D)(y_A - y_B) = 0\\
per(A,B,C,D) &\equiv (x_A - x_B)(x_C - x_D) + (y_A - y_B)(y_C - y_D) = 0
\end{aligned}$$

$$eq\_distance(A,B,C,D) \equiv$$

$$(x_A - x_B)^2 + (y_A - y_B)^2 - (x_C - x_D)^2 - (y_C - y_D)^2 = 0$$

$$eq\_angle(A,B,C,D,E,F) \equiv$$

$$\begin{aligned}
&((y_B - y_A) * (x_B - x_C) - (y_B - y_C) * (x_B - x_A)) *\\
&((x_E - x_D) * (x_E - x_F) + (y_E - y_D) * (y_E - y_F))\\
&=\\
&((y_E - y_D) * (x_E - x_F) - (y_E - y_F) * (x_E - x_D)) *\\
&((x_B - x_A) * (x_B - x_C) + (y_B - y_A) * (y_B - y_C))
\end{aligned}$$

3.1.1 Translating a construction into a statement for ATP.

We need to translate from one language to the other one. The idea of the translation consist in maintaining the invariant that lines and circles are always defined by two points. Of course this is not true in GeoProof. For instance one can build a line as the parallel of another line passing through a point. In such a case we need to define a second defining point for the line. For that purpose we generate new points during the translation. We define the translation by case distinction on the construction. Table 1 gives the defining points for each line and circle depending on how these objects have been constructed. $P1_l$,$P2_l$ and O_c are fresh variables. For each line and circle we associate some fresh variables. These new variables which do not appear in the original figure are used to define lines and circles when we do not have two points on the object on the figure we translate from.

Lines are defined by two points $\mathcal{P}_1(l)$ and $\mathcal{P}_2(l)$. When we already know at least one of the defining points we use it instead of creating a new point because it simplifies the generated formulas.

Circles are defined by their center $\mathcal{O}(c)$ and a point $\mathcal{P}(c)$ on the circle.

Table 2 provides the translation of GeoProof constructions[5] into the language accepted by the embedded theorem prover. Incidentally, it gives a subset of the constructions of the language of GeoProof. The non degeneracy conditions are inspired by those in [12]. The predicate *isotropic* is defined by:

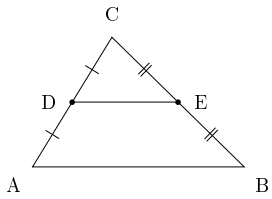
$$isotropic(A, B) \equiv perpendicular(A, B, A, B)$$

In Euclidean geometry it is equivalent to $A = B$ but not in metric geometry. We produce a statement which is interpreted in the metric geometry because Wu and Gröbner bases methods are complete only for metric geometry. For more information about this see [12,11]. Moreover if I_1 and I_2 are the two intersections of a circle and of a line or a circle then we add the fact that $I_1 \neq I_2$ in the hypotheses. Note that different constructions of the same figure can lead to different degeneracy conditions and hence different formulas.

3.1.2 An example

Let's take the midpoint theorem as an example, it states that:

Theorem 3.1 (midpoint) *Let ABC be a triangle, and let D and E be the midpoints of AC and BC respectively. Then the line DE is parallel to the base AB.*



The construction is translated into the following statement:

```
(((((is_midpoint(D,C,A) /\ is_midpoint(E,C,B))/\
~C=A) /\ ~A=B) /\ ~B=C) /\ ~D=E) /\ ~A=B
```

The fact that $AB \parallel DE$ is then checked using the Gröbner basis method.

[5] To simplify the presentation we only provide the translation for the main GeoProof constructions.

GeoProof Construction	Defining points
l passing through A and B	$\mathcal{P}_1(l) = A\ \mathcal{P}_2(l) = B$
l parallel line to m passing through A	$\mathcal{P}_1(l) = A\ \mathcal{P}_2(l) = P2_l$
l perpendicular line to m passing through A	$\mathcal{P}_1(l) = A\ \mathcal{P}_2(l) = P2_l$
l perpendicular bisector of A and B	$\mathcal{P}_1(l) = P1_l\ \mathcal{P}_2(l) = P2_l$
l bisector of the angle formed by A, B and C	$\mathcal{P}_1(l) = B\ \mathcal{P}_2(l) = P2_l$
c circle of center O passing through A	$\mathcal{O}(c) = O\ \mathcal{P}(c) = A$
c circle passing through A,B and C	$\mathcal{O}(c) = O_c\ \mathcal{P}(c) = A$
c circle whose diameter is A B	$\mathcal{O}(c) = O_c\ \mathcal{P}(c) = A$

Table 1
Definition of the defining points of circles and lines

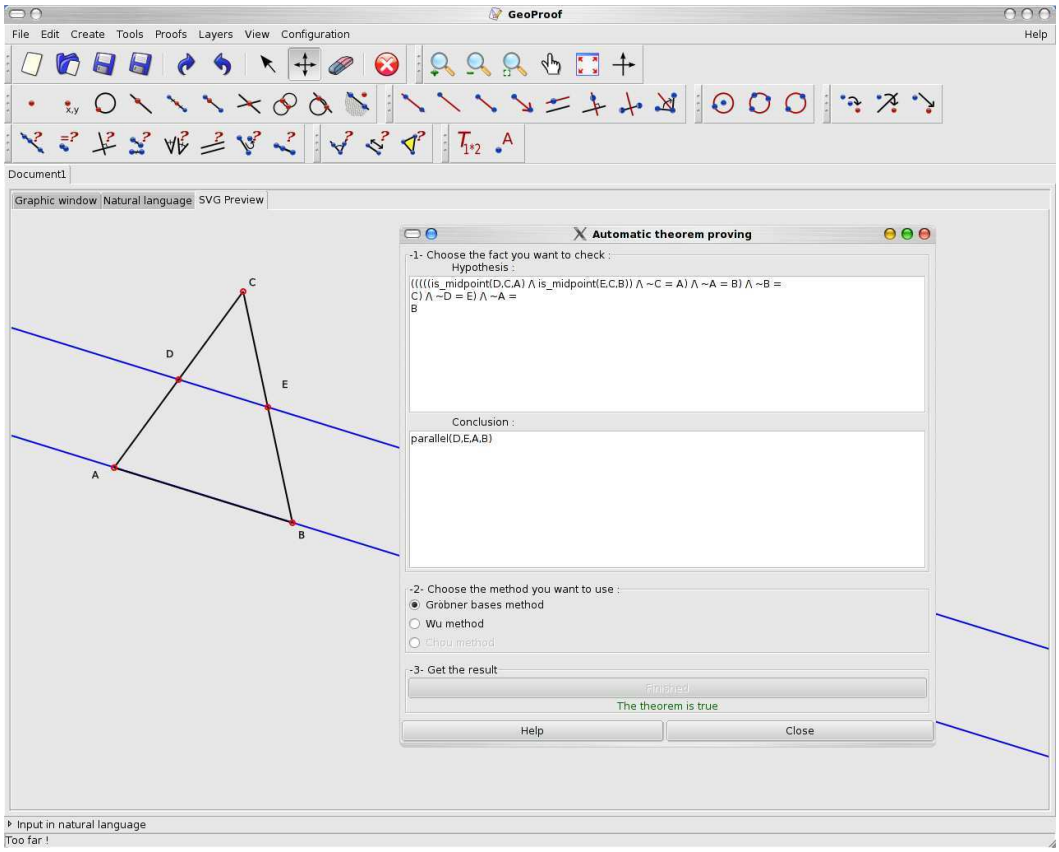


Fig. 2. Checking the midpoint theorem using the embedded theorem prover.

3.1.3 Dealing with non-degeneracy conditions

Non degeneracy conditions play a crucial role in formal geometry, this has been emphasized by most papers about formalization of geometry [20,28,30]. This translation is not an exception, we must be careful about the semantics of the generated statements. For this translation we have decided to consider GeoProof as a tool

GeoProof Construction	**Predicate form**
Free point	$true$
Point P on line l	$collinear(P, \mathcal{P}_1(l), \mathcal{P}_2(l))$
Point P on circle c	$\mathcal{O}(c)\mathcal{P}(c) = P\mathcal{O}(c)$
I midpoint of A and B	$IA = IB \wedge collinear(I, A, B)$
I intersection of l_1 and l_2	$collinear(I, \mathcal{P}_1(l_1), \mathcal{P}_2(l_1)) \wedge$ $collinear(I, \mathcal{P}_1(l_2), \mathcal{P}_2(l_2)) \wedge$ $\neg parallel(\mathcal{P}_1(l_1), \mathcal{P}_2(l_1), \mathcal{P}_1(l_2), \mathcal{P}_2(l_2))$
I an intersection of c_1 and c_2	$I\mathcal{O}(c_1) = \mathcal{O}(c_1)\mathcal{P}(c_1) \wedge$ $I\mathcal{O}(c_2) = \mathcal{O}(c_2)\mathcal{P}(c_2) \wedge$ $\neg isotropic(\mathcal{O}(c_1), \mathcal{O}(c_2))$
I an intersection of c and l	$I\mathcal{O}(c) = \mathcal{O}(c)\mathcal{P}(c) \wedge$ $collinear(I, \mathcal{P}_1(l), \mathcal{P}_2(l)) \wedge$ $\neg isotropic(\mathcal{P}_1(l), \mathcal{P}_2(l))$
l passing through A and B	$A \neq B$
l parallel to m passing through A	$parallel(A, \mathcal{P}_2(l), \mathcal{P}_1(m), \mathcal{P}_2(m)) \wedge$ $A \neq \mathcal{P}_2(l)$
l perpendicular to m passing through A	$perpendicular(A, \mathcal{P}_2(l), \mathcal{P}_1(m), \mathcal{P}_2(m)) \wedge$ $A \neq \mathcal{P}_2(l)$
l perpendicular bisector of A and B	$\mathcal{P}_1(l)A = \mathcal{P}_1(l)B \wedge \mathcal{P}_2(l)A = \mathcal{P}_2(l)B \wedge$ $\mathcal{P}_1(l) \neq \mathcal{P}_2(l) \wedge A \neq B$
l bisector of the angle A,B,C	$eq\_angle(A, B, \mathcal{P}_2(l), \mathcal{P}_2(l), B, C) \wedge$ $B \neq \mathcal{P}_2(l) \wedge A \neq B \wedge B \neq C$
c circle of center O passing through A	$true$
c circle whose diameter is A B	$collinear(\mathcal{O}(c), A, B) \wedge$ $\mathcal{O}(c)A = \mathcal{O}(c)B$

Table 2
Predicate form for each type of construction

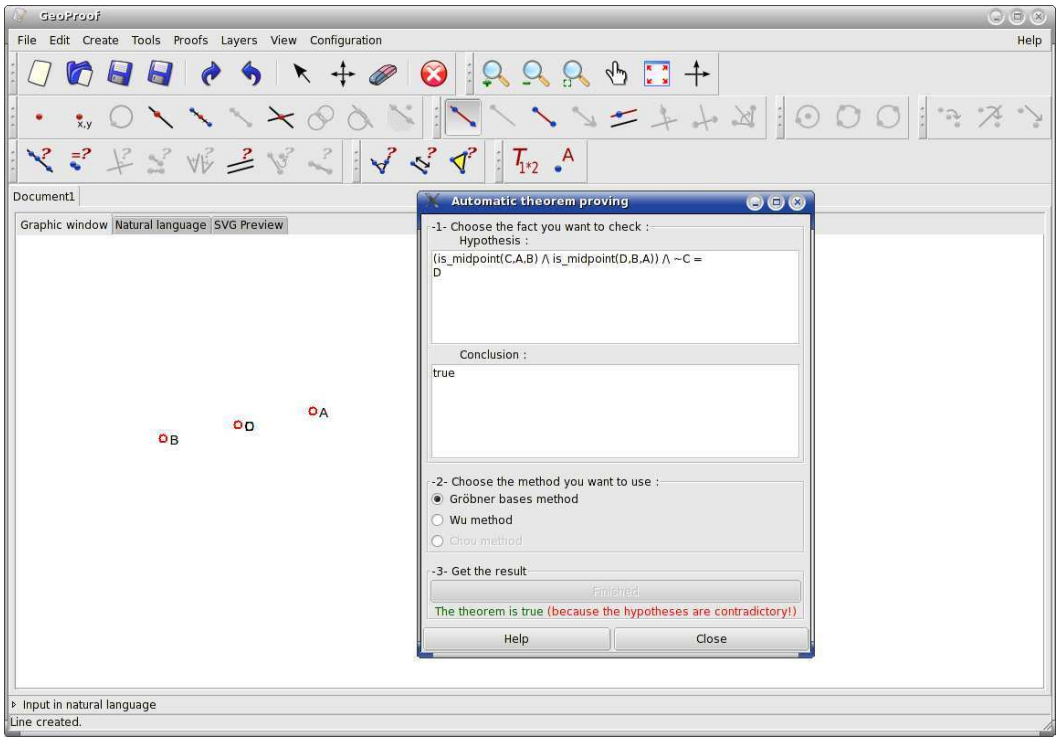


Fig. 3. Trying to prove a property with contradictory hypotheses.

which permits to define a geometric *formula* and it does not build a *model* of this formula. The user can define "impossible" figures. For instance if we perform the following construction:
First, create two points A and B and then create the midpoint C of the segment $[AB]$ and the midpoint D of the segment $[BA]$. Finally, create the line passing through C and D. Then if we try to prove that $C \neq D$, GeoProof should answer "yes", as the hypotheses of the theorem are inconsistent (*ex falso quod libet*). This is consistent with logic but not with the user's intuition because the "impossible" objects are not displayed by GeoProof. This is why in fact we need to check first if we can prove *false*, if this is the case we can warn the user that its construction is impossible as shown on Figure 3. Note that on the example shown we have not created exactly the line passing through A and A, because GeoProof does not allow this particular degenerated construction. We have created two points which are equal (C and D) using the midpoint construction applied twice to the same segment.

3.2 Using Coq

In [30] we have described the implementation of Chou, Gao and Zhang's decision procedure for affine geometry in the Coq proof assistant. Here we want to export a construction built using GeoProof into a statement in the language of the Coq development. Our implementation of Chou, Gao and Zhang's decision procedure is restricted to affine plane geometry. Hence in GeoProof the tools which do not have any corresponding concept in the Coq implementation are greyed out. The Coq development is based on the axiom system shown on Table 3. To ease the Coq

Points	$\text{Point} : \text{Set}$
Field	F is a field $2 \neq 0$
Signed distance	$\overline{\ \ } : \text{Point} \to \text{Point} \to F$ $\overline{AB} = 0 \iff A = B$
Signed area	$\mathcal{S} : \text{Point} \to \text{Point} \to \text{Point} \to F$ $\mathcal{S}_{ABC} = \mathcal{S}_{CAB}$ $\mathcal{S}_{ABC} = -\mathcal{S}_{BAC}$
Chasles'axiom	$\mathcal{S}_{ABC} = 0 \to \overline{AB} + \overline{BC} = \overline{AC}$
Dimension	$\exists\, A, B, C : \text{Point}, \mathcal{S}_{ABC} \neq 0$ $\mathcal{S}_{ABC} = \mathcal{S}_{DBC} + \mathcal{S}_{ADC} + \mathcal{S}_{ABD}$
Construction	$\forall r : F\ \exists P : Point, \mathcal{S}_{ABP} = 0 \wedge \overline{AP} = r\overline{AB}$ $A \neq B \wedge \mathcal{S}_{ABP} = 0 \wedge \overline{AP} = r\overline{AB} \wedge \mathcal{S}_{ABP'} = 0 \wedge \overline{AP'} = r\overline{AB} \to P = P'$
Proportions	$A \neq C \to \mathcal{S}_{PAC} \neq 0 \to \mathcal{S}_{ABC} = 0 \to \frac{\overline{AB}}{\overline{AC}} = \frac{\mathcal{S}_{PAB}}{\mathcal{S}_{PAC}}$

Table 3
The Chou axiom system (slightly modified for the formalization in Coq).

formalization, this axiom system has been slightly modified compared to the axiom system found in [13]. In the original axiom system the ratio of two oriented distances $\frac{\overline{AB}}{\overline{CD}}$ is defined only when AB is parallel to CD. Here we do not put this restriction at the axiom system level but only when we state theorems involving ratios. It is clear that this axiom system is based on points. Hence we have to perform a translation similar to those described in the last section. Table 4 gives the translation of some common geometric notions in the language of the axiom system. Figure 4 shows the translation of the statement corresponding to the midpoint theorem in the syntax of Coq.

Geometric notions	Formalization
A,B and C are collinear	$\mathcal{S}_{ABC} = 0$
$AB \parallel CD$	$\mathcal{S}_{ABC} = \mathcal{S}_{ABD}$
I is the midpoint of AB	$\frac{\overline{AB}}{\overline{AI}} = 2 \wedge \mathcal{S}_{ABI} = 0$

Table 4
Expressing some common geometric notions using $\mathcal{S}$ and ratios

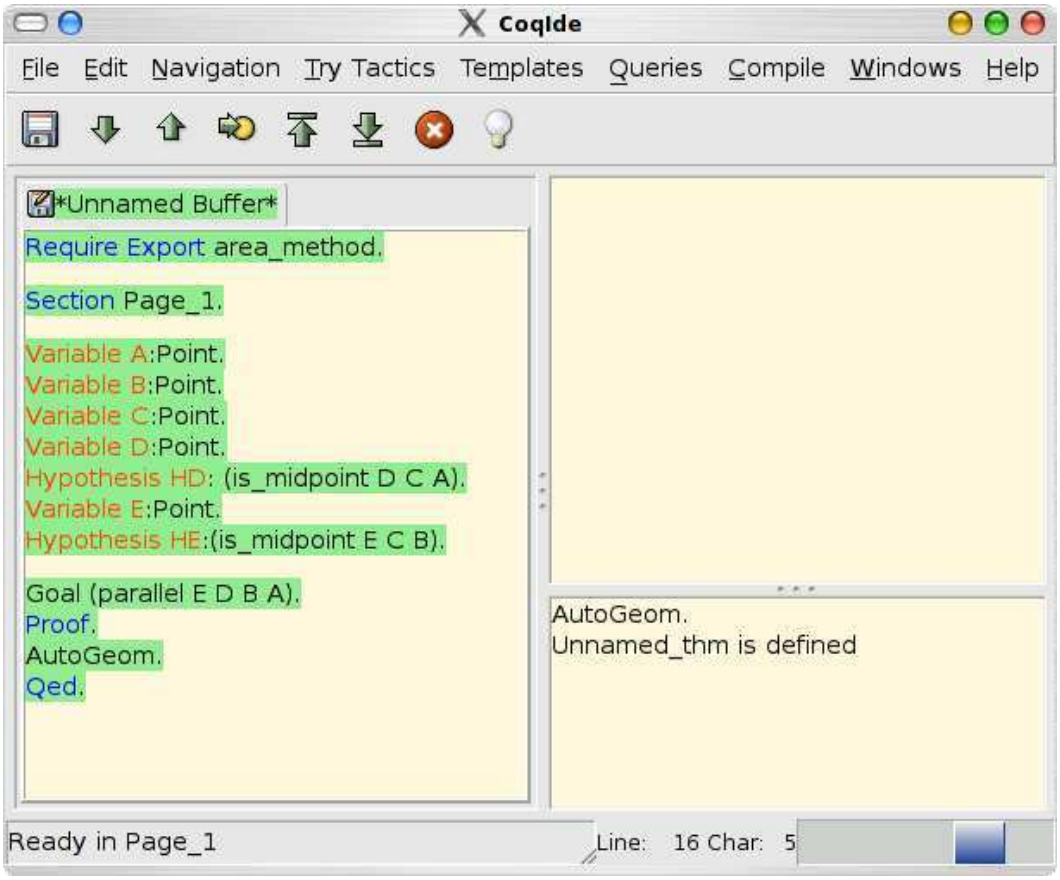
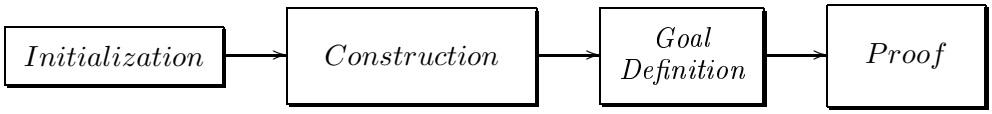


Fig. 4. The midpoint theorem, expressed in the Coq language for Chou decision procedure.

4 Interactive input

In this section we describe the interactive proof mode of GeoProof. Thanks to the configuration menu, the user can choose between two interactive modes, the first one uses the language described in section 3.2 and the second one uses the language of the Coq development for high school geometry by Frédérique Guilhot [20]. In the first mode the user can deal with affine plane geometry and in the second mode with Euclidean plane geometry. The interaction with Coq is performed through the CoqIDE user interface. GeoProof communicates with CoqIDE [6] thanks to a private clipboard. We have started by implementing the translation from a GeoProof construction to a Coq statement. We perform the same translation as in [9] except that it is in the reverse direction (here we translate *to* Coq) [7] .

The interactive mode of GeoProof is decomposed in four steps:



In the initialization phase, the communication between CoqIDE and GeoProof is started. Depending on the language used some construction tools which can not be exported to Coq are greyed out in GeoProof. The Coq definitions corresponding to the language used are loaded using the Coq command `Require`. A new section is opened. If the user had already constructed some objects before starting the interactive proof mode, these objects are now exported to Coq. Objects which do not have any meaning in the language selected are ignored.

In the construction phase the objects created by the user are added in the Coq context with their corresponding assumptions. On the example shown [8] on Figure 8 this corresponds to the `Variable` and `Hypothesis` commands.

In the goal phase the user needs to define what he wants to prove.
In the context of education this phase can be presented as an exercise consisting in finding an interesting conjecture about the figure. For that purpose GeoProof provides several features:

(i) The user can move the free points of the figure to guess the invariants.

(ii) When the user has guessed a conjecture, he can make a first experiment to check the conjecture by building a dynamic label to perform mesures on the figure.
A dynamic label is a text element enriched with the possibility to display the result of a computation defined using a small language ([32]). Thanks to a configuration file the user can choose at which precision (which may be arbitrary large) the computations are performed. If the mathematical expressions contained in the text elements depend on other points of the figure, the text is updated in real time when the user change the position of the free points.

[6] This feature requires CoqIDE version 8.1 or later.

[7] In the future we should merge our developments to allow communication in both directions, this requires a more complex communication system as explained in the future work section.

[8] The predicates names are in French because this development is focused on the French high-school curriculum

Fig. 5. The definition of a dynamic label.

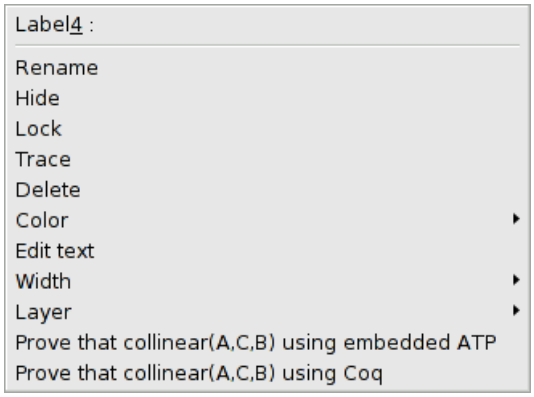


Fig. 6. The contextual menu associated to a dynamic label.

The dynamic part of the labels can contain measures and predicate tests using variables depending on other objects. Figure 5 shows an example of a dynamic label to test if three points are collinear. Using predefined dynamic labels the user can check easily for example if two lines are parallel (on the specific instance of the figure displayed). Then if he wants to prove the fact represented by the label, he can right click on the label and choose the corresponding menu entry. Figure 6 shows the contextual menu of a dynamic label.

In the proof phase the user proves his statement within CoqIDE. Hence, the current implemantation of GeoProof requires to know how to use Coq. This will

```
Ltac DecompEx H P := elim H;intro P;intro;clear H.

Ltac let_intersection I A B C D :=
let id1 := fresh in ((assert (id1:exists I,
I =  pt_intersection (line A B) (line C D));
[apply (existence_pt_intersection)|DecompEx id1 I])).
```

Fig. 7. The tactic to prove the existence of the point of intersection.

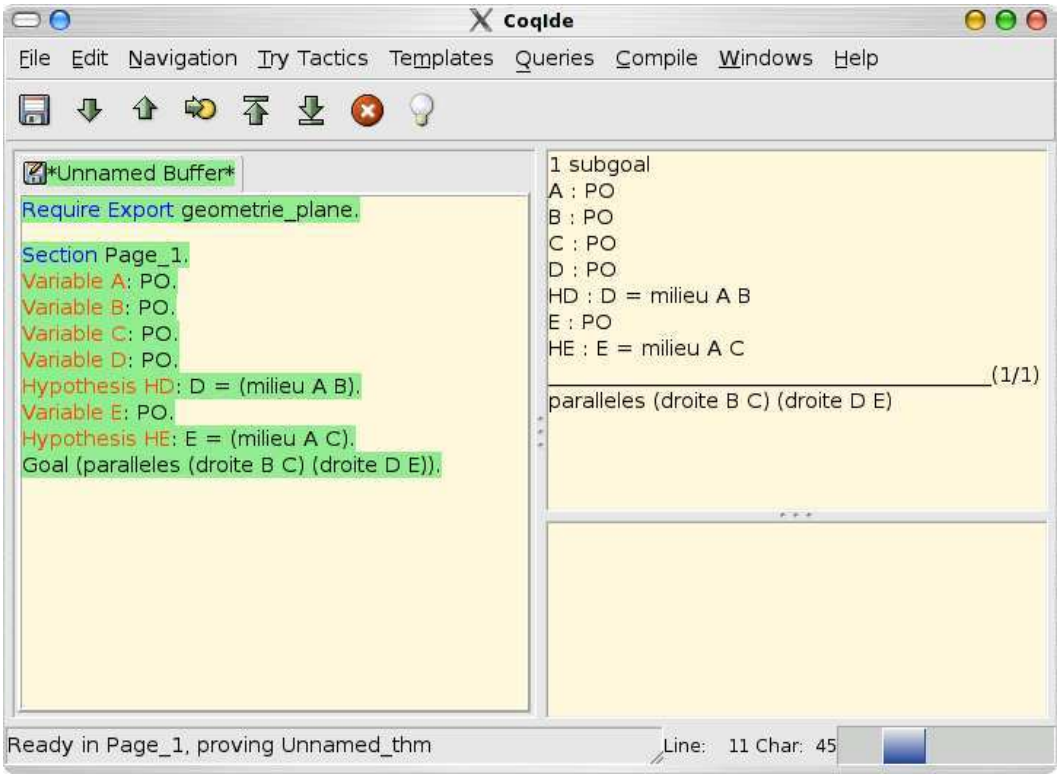


Fig. 8. The midpoint theorem in the language used by Frédérique Guilhot's Coq development.

be improved in future versions by adding some features to allow the application of theorems within GeoProof. If during the proof a new object needs to be created, he can do it using GeoProof. Indeed when a new object is added in GeoProof a Coq tactic is pasted into CoqIDE. This tactic applies the theorem which proves the existence of the object which has just been created and introduce in the context the knowledge about this new object. In some cases this generates non-degeneracy conditions which need to be proved by the user. Figure 7 shows the command (defined in Ltac - the tactic language of Coq) which is used when the user creates a point at the intersection of two lines.
If the user deletes an object in GeoProof it is removed from the Coq context thanks to the `clear` command of Coq. If the user wants to delete some object without deleting it in Coq, he can hide the object in GeoProof.

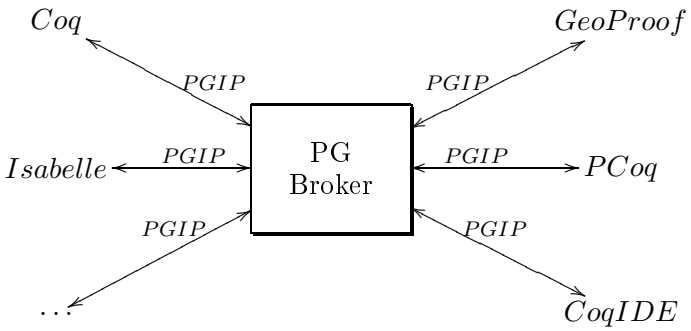


Fig. 9. Integrating GeoProof in the proof general infrastructure

5 Future Work

The current prototype of GeoProof uses a private clipboard [9] as a communication pipe between GeoProof and the Coq Interactive Development Environment. This approach has the advantage to be both easy to implement and easy to use. The user can start the interaction without any configuration step, he just needs to launch GeoProof and CoqIDE on the same computer. But this infrastructure has some limitations. First, the communication with Coq is done using the Coq syntax, which is easy to produce but hard to parse. Second, the synchronization between what is typed in CoqIDE and the input generated by GeoProof is not ensured. A better infrastructure for the communication between Coq and GeoProof would be to use the Proof General Interaction Protocol (PGIP) framework [39,4]. This framework is based on XML and allow to have several interfaces interacting at the same time with one proof assistant. This is exactly what we need because as mentioned before, some proofs are easier to grasp diagrammatically and some are better presented the classic way (proofs using complex numbers for instance). In our example, GeoProof and CoqIDE would interact with the Coq proof assistant. But this could be generalized to other proof assistants and graphical user interfaces such as Isabelle, Eclipse/Proof General and PCoq as shown on Figure 9. This approach would require implementation of PGIP within Coq, CoqIDE and GeoProof.

The proving features of GeoProof in itself should also be extended. We need to add the possibility to apply a theorem diagrammatically by drag and drop and to mark facts on the diagram to produce new assertions in Coq. We could also transform macro constructions into proof of existence of geometric objects verifying some properties.

Another planned extension of GeoProof is to adapt it to deal with diagrammatic proofs in abstract term rewriting (see the first chapter of [5]). We have formalized in [31] the kind of diagrams which are usually found in the rewriting literature. The next step is to implement this formalization in GeoProof to provide a high level input language for proofs in abstract rewriting. The design presented in this paper can be adapted to abstract term rewriting.

[9] Technically, we use a feature provided by GTK: we create a clipboard identified by a name (here "`GeoProof`") which is different from the standard clipboard.

6 Conclusion

Proof is a crucial aspect of mathematics and hence must have a prominent role in the education. The most widely used software in the teaching of mathematics are mainly used to explore, visualize, calculate, find counter examples, conjectures, or check facts, but most of them can not be used to build a proof in itself. We believe that proof assistants should be adapted to fulfill this need.

We have presented in the paper a prototype which aims at integrating dynamic geometry, automatic theorem proving and formal proof. This should be considered as a first step toward the use of a proof assistant in the classroom.

Availability

GeoProof is available at: http://home.gna.org/geoproof/

Acknowledgements

I want to thank Hugo Herbelin for his help during the elaboration of this work and Frédérique Guilhot for her comments and the formal proofs she has added to her development for GeoProof.

References

[1] Ag-Almouloud. *L'ordinateur, outil d'aide à l'apprentissage de la démonstration et de traitement de données didactiques*. PhD thesis, Université de Rennes, 1992.

[2] Ahmed Amerkad, Yves Bertot, Loïc Pottier, and Laurence Rideau. Mathematics and proof presentation in Pcoq. In *Workshop Proof Transformation and Presentation and Proof Complexities in connection with IJCAR 2001*, June 2001.

[3] John R. Anderson, C. F. Boyle, and Gregg Yost. The geometry Tutor. In *IJCAI Proceedings*, pages 1–7, 1985.

[4] David Aspinall, Christoph Lüth, and Daniel Winterstein. A framework for interactive proof. Technical report, 2004.

[5] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, USA, 1998.

[6] Nicolas Balacheff, Sylvie Pesty, Michel Occello, Ricardo Caffera, Carine Webber, and Nicolas Peltier. Baghera, 1999-2002. http://www-baghera.imag.fr.

[7] Jon Barwise and Gerard Allwein, editors. *Logical Reasoning with Diagrams*. Oxford University Press, 1996.

[8] Philippe Bernat. Chypre: Un logiciel d'aide au raisonnement. Technical Report 10, IREM, 1993.

[9] Yves Bertot, Frédérique Guilhot, and Loïc Pottier. Visualizing geometrical statements with GeoView. *Electronic Notes in Theoretical Computer Science*, 103:49–65, September 2003.

[10] Yves Bertot and Laurent Thery. A generic approach to building user interfaces for theorem provers. *The Journal of Symbolic Computation*, 25:161–194, 1998.

[11] Shang-Ching Chou. *Mechanical Geometry Theorem Proving*. D. Reidel Publishing Company, 1988.

[12] Shang-Ching Chou and Xiao-Shan Gao. A class of geometry statements of constructive type and geometry theorem proving. In *Proceeding of CADE 92*, 1992.

[13] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. *Machine Proofs in Geometry*. World Scientific, Singapore, 1994.

[14] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. Automated generation of readable proofs with geometric invariants, theorem proving with full angle. *Journal of Automated Reasoning*, 17:325–347, 1996.

[15] Coq development team, The. *The Coq proof assistant reference manual, Version 8.0*. LogiCal Project, 2004.

[16] Fulvia Furinghetti and Paoloa Domingo. To produce conjectures and to prove them within a dynamic geometry environment: a case study. In *Proceeding of Psychology of Mathematics 27th international Conference*, pages 397–404, 2003.

[17] Xian-Shan Gao and Qiang Lin. MMP/Geometer - a software package for automated geometry reasoning. In F. Winkler, editor, *Proceedings of ADG 2002*, Lecture Notes in Computer Science, pages 44–46. Springer-Verlag, 2002.

[18] Xiao-Shan Gao. Geometry expert, software package, 2000. http://www.mmrc.iss.ac.cn/~xgao/gex.html.

[19] Jacques Gressier. Geometrix, 1988-1998. http://perso.wanadoo.fr/jgressier/ENGLISH/english.html.

[20] Frédérique Guilhot. Formalisation en coq et visualisation d'un cours de géométrie pour le lycée. *Revue des Sciences et Technologies de l'Information, Technique et Science Informatiques, Langages applicatifs*, 24:1113–1138, 2005. Lavoisier.

[21] John Harrison. Automatic theorem proving examples, 2003. http://www.cl.cam.ac.uk/users/jrh/atp/index.html.

[22] Nicholas Jackiw. The geometer's sketchpad, 1990. http://www.keypress.com/.

[23] Mateja Jamnik. *Mathematical Reasoning with Diagrams: From Intuition to Automation*. CSLI Press, 2001.

[24] Ulrich Kortenkamp. *Foundations of Dynamic Geometry*. PhD thesis, ETH Zürich, 1999.

[25] Ulrich Kortenkamp and Jürgen Richter-Gebert. Using automatic theorem proving to improve the usability of geometry software. In *Mathematical User Interface*, 2004.

[26] Jean-Marie Laborde and Franck Bellemain. Cabri-geometry II, 1993-1998. http://www.cabri.net.

[27] Vanda Luengo. *Cabri-Euclide: Un micromonde de Preuve intégrant la réfutation.* PhD thesis, Université Joseph Fourier, 1997.

[28] Laura Meikle and Jacques Fleuriot. Formalizing Hilbert's Grundlagen in Isabelle/Isar. In *Theorem Proving in Higher Order Logics*, pages 319–334, 2003.

[29] Nathaniel Miller. *A diagrammatic formal system for Euclidean geometry.* PhD thesis, Cornell University, May 2001.

[30] Julien Narboux. A decision procedure for geometry in Coq. In Slind Konrad, Bunker Annett, and Gopalakrishnan Ganesh, editors, *Proceedings of TPHOLs'2004*, volume 3223 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[31] Julien Narboux. A formalization of diagrammatic proofs in abstract rewriting. 2006.

[32] Julien Narboux. *The user manual of GeoProof*, July 2006.

[33] Dominique Py. *Reconnaissance de plan pour l'aide à la démonstration dans un tuteur intelligent de la géométrie.* PhD thesis, Université de Rennes, 1990.

[34] Jürgen Richter-Gebert and Ulrich Kortenkamp. Die interaktive geometrie software cinderella book and cd-rom. German school-edition of the Cinderella software, 1999. http://cinderella.de.

[35] Jacob T. Schwartz. Probabilistic algorithms for verification of polynomial identities. In *Symbolic and algebraic computation*, volume 72 of *Lecture Notes in Computer Science*, pages 200–215, Marseille, 1979. Springer-Verlag.

[36] Sean Wilson and Jacques Fleuriot. Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs. In *ETAPS Satellite Workshop on User Interfaces for Theorem Provers (UITP)*, Edinburgh, 2005.

[37] Daniel Winterstein. Dr.Doodle: A diagrammatic theorem prover. In *Proceedings of IJCAR 2004*, 2004.

[38] Daniel Winterstein. *Using Diagrammatic Reasoning for Theorem Proving in Continous Domain.* PhD thesis, The University of Edinburgh, 2004.

[39] Daniel Winterstein, David Aspinall, and Christoph Lüth. PG/Eclipse: A generic interface for interactive proof. Technical report, 2004.

[40] Wen-Tsün Wu. On the decision problem and the mechanization of theorem proving in elementary geometry. In *Scientia Sinica*, volume 21, pages 157–179. 1978.

[41] Oleksiy Yevdokimov. About a constructivist approach for stimulating students' thinking to produce conjectures and their proving in active learning of geometry. In *Fourth Congress of the European Society for Research in Mathematics Education*, 2004.

GeoThms — a Web System for Euclidean Constructive Geometry

Pedro Quaresma[1,2]

CISUC/Department of Mathematics
University of Coimbra
3001-454 Coimbra, Portugal

Predrag Janičić[3,4]

Faculty of Mathematics, University of Belgrade
Studentski trg 16, 11000 Belgrade, Serbia & Montenegro

Abstract

GeoThms is a web-based framework for exploring geometrical knowledge that integrates Dynamic Geometry Software (DGS), Automatic Theorem Provers (ATP), and a repository of geometrical constructions, figures and proofs. The GeoThms users can easily use/browse through existing geometrical content and build new contents. In this paper we describe GeoThms functionalities, focusing on the interface solutions required for a system aimed at supporting studying and teaching geometry via Internet. GeoThms is a publicly accessible system with a growing body of geometrical constructions and formally proven geometrical theorems. We believe that, with the help of all its users it will become an important Internet resource for geometry.

Keywords: Web interfaces for proof systems, automated geometry theorem proving, dynamic geometry software.

1 Introduction

Our motivation is to build and maintain a publicly accessible and widely used Internet based framework for constructive geometry. It should be used for teaching and studying geometry, but also as a major Internet repository for geometrical constructions. We have built a system, GeoThms, that links Dynamic Geometry Software (DGS), Automatic Theorem Provers (ATP), and GeoDB, a database of geometrical constructions, figures and proofs. The DGSs currently used within GeoThms

[1] This work was partially supported by programme POSC.

[2] Email: pedro@mat.uc.pt

[3] This work was partially supported by Serbian Ministry of Science and Technology grant 144030. Also, partially supported by the programme POSC, by the Centro International de Matemtica (CIM), under the programme "Research in Pairs", while visiting Coimbra University under the Coimbra Group Hospitality Scheme.

[4] Email: janicic@matf.bg.ac.yu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

are GCLC [7,12] and Eukleides [16,18], two widely used dynamic geometry packages. The ATP used, GCLCprover [13,17], is based on the area method [4,6,15,17], and it produces human readable, synthetic geometrical proofs. GeoThms provides a web workbench that tightly integrates mentioned tools into a single framework for constructive geometry. The web interface is a server-side solution written in PHP, designed to enable GeoThms users to easily browse through the list of geometric problems, their statements, illustrations and proofs, and also to interactively use the drawing and automatic proof tools. GeoThms is accessible at http://hilbert.mat.uc.pt/~geothms.

There are several systems related to GeoThms. Some of them combine features of DGS and automated theorem provers, some of them have web interfaces, and some of them provide repositories of geometrical theorems. We are not aware of any system that, like GeoThms, gives full, web-based access to DGS, use theorem proving with human-readable proofs generated and provides open repository of geometrical constructions and conjectures. Section 6 gives more details about related work.

Paper overview.

Section 2 describe GeoThms components; Section 3 presents the structure of the web interface; Section 4 is about communication and representation issues; Section 5 presents GeoThms through some illustrated examples. Section 6 discuss related work. Section 7 discusses further work, and Section 8 draws final conclusions.

2 Framework Components

GeoThms, is a framework that links dynamic geometry software, geometry automatic theorem provers, and a repository of geometry problems providing a common web interface for all these tools (see Figure 1). In this section, we give a brief description of the tools that are currently integrated in GeoThms:

GCLC and Eukleides

GCLC[5] [7,12] and Eukleides[6] [16,18] are two DGSs. They both use declarative languages to specify geometrical constructions. Hence, in using these tools, producing mathematical illustrations is based on "describing figures" rather than on "drawing figures". These descriptions directly reflect meaning of mathematical objects to be presented, and are easily understandable to mathematicians. Both tools have graphical user interfaces and produce, in LaTeX form, illustrations that correspond to geometric constructions.

GCLCprover

GCLCprover is an ATP based on the area method [4,6,15]. It allows formal deductive reasoning about objects constructed within DGSs. It produces proofs that

[5] GCLC package is freely available from `www.matf.bg.ac.yu/~janicic/gclc/`. The mirrored version is available from EMIS (The European Mathematical Information Service) `www.emis.de/misc/index.html`. There are versions of GCLC for Windows and for Linux.

[6] Eukleides is available from `http://www.eukleides.org`, There are versions for a number of languages. The first author of this paper is responsible for the Portuguese version of Eukleides: EukleidesPT is available from `http://gentzen.mat.uc.pt/~EukleidesPT/`

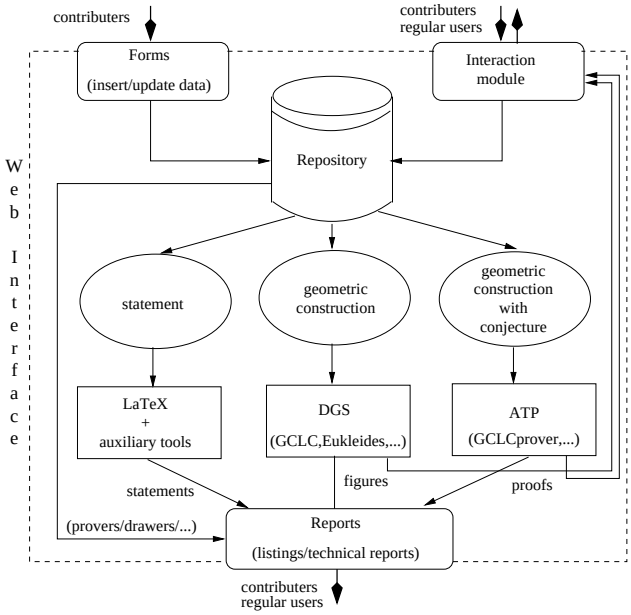


Fig. 1. The GeoThms Framework

are human-readable (in LaTeX and XML formats), and with an explicit justification for every proof step. GCLCprover is tightly integrated with GCLC, so one can use the prover to reason about a GCLC construction, without changing and adapting it for the deductive process, the users only need to add a conclusion they want to prove. The geometrical constructions made within GCLC are internally transformed into primitive constructions of the area method, and in some cases, some auxiliary points are introduced. We have developed a tool `euktogclcprover`, that converts Eukleides files to GCLCprover files, allowing the prover to be used with geometric constructions described within Eukleides.

The geoDB database

geoDB keeps geometric constructions, illustrations, conjectures, and proofs. Figure 2 shows the structure of the database. The main entities of the database are: *figures*, descriptions of geometrical constructions; *theorems*, statements of theorems, written in LaTeX form; *proofs*, geometrical constructions with conjectures.

Geometrical constructions are described and stored in the database in declarative languages of dynamic geometry tools such as GCLC and Eukleides, and in a common XML format. Figures are generated directly on the basis of geometric specifications, by GCLC and Eukleides and stored as JPEG files and SVG files. Conjectures are described and stored in a form that extends descriptions of geometrical constructions. The specifications of conjectures are used (directly or via a converter) by GCLCprover. Proofs are generated by GCLCprover and stored as XML files (rendered by XSLT, using a layout specified by `GeoCons_proof.xsl`) and as PDF files (produced by LaTeX, using a layout specified by `gclc_proof.sty`). A geometric theorem can have more than one figure and/or more than one proof, made by different tools and made by different users.

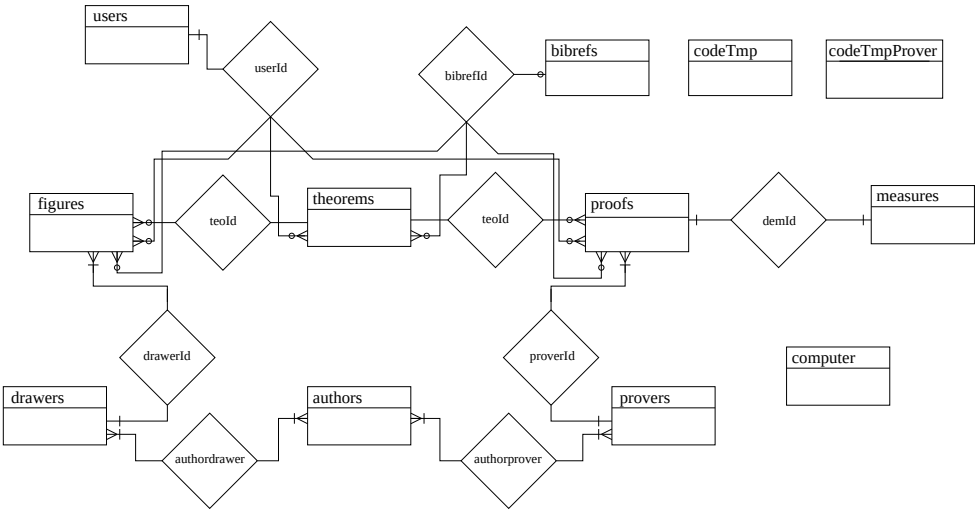


Fig. 2. geoDB — Entity-relationship Diagram

The database also has the following auxiliary entities: *bibrefs*, bibliographic references, in BibTeX format; *drawers & provers*, information about the available programs; *authors*, information about the authors; *users*, information about registered users; *computer*, information about computers used as the test benches. The tables `codeTmp` and `codeTmpProver` are used to store temporary information, deleted after each session, for the interactive section of GeoThms.

3 The Web Interface General Structure

The structure of the web interface has two main levels (see Figure 3).

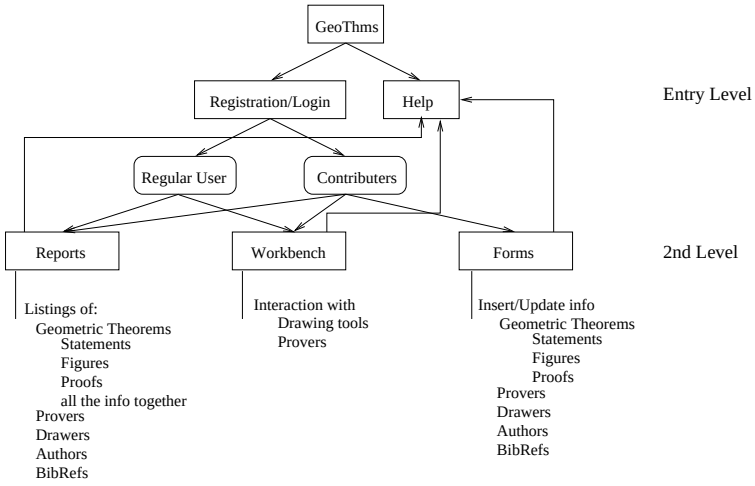


Fig. 3. GeoThms — Web Interface

The entry level (see Figure 4), accessible to all web-users, has some basic information about GeoThms, including documents about the GCLCprover and the area method. There is also an entry point to the GeoThms forums. This level offers registering options, and it gives access to other levels of the system. Regular users have access to the second level, where they can browse the data from the database

(in a formatted, or in a plain textual form) and use the drawing/proving programs for interactive work. Regular users can apply for the *contributor* status with which they can also insert new data and can update the data inserted previously in the database. There is also an administration level, invisible to the user. It is used to change the status of the users, and other administrative tasks.

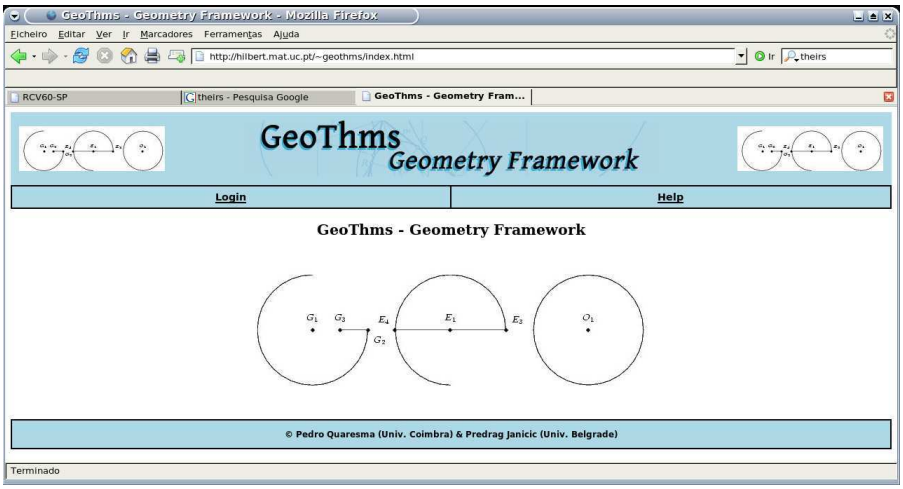


Fig. 4. GeoThms — Home Page

4 Communication and Representation Issues

GeoThms is a server side web system with integrated DGSs and ATPs tools. It is not oriented to some particular browser and/or operating system. As a web service GeoThms emphasise:

- a simple interface, based on using geometrical specification languages of the underlying geometrical tools;
- a low communication burden.

To achieve these goals we decided to use a server side Apache/PHP/MySQL solutions, and standard features of HTML language to deal with input/output.

A basic communication, concerning describing geometrical constructions and conjectures, is based on formal languages of the underlying geometrical tools. Despite some good features of point-and-click-based descriptions of constructions, we believe that in this context, communication based on textual descriptions is a better solution. For instance, this approach enables full access to the underlying systems, different geometrical tools can be uniformly integrated, stresses the fact that geometrical constructions are formal procedures, etc. Notice that both DGSs tools currently supported (GCLC and Eukleides) provide also graphical interfaces, so it is possible for a user to use these tools' rich graphical interfaces locally and then transfer the results to GeoThms.

Concerning internal representation of data, within GeoThms, descriptions of constructions and conjecture are stored as GCLC code, as Eukleides code, or in XML form. There are tools for converting between these formats, while XML format has the central position, as an interchange format. When adding new geometrical tools,

it will be sufficient to develop converters from its format to XML and vice versa. This enables converting from any format to any other, and consequently makes usable the whole of the repository to any geometrical tool. Figures are stored in JPEG format, but also in SVG format.

Within GeoThms, data are presented in:

textual form with the following choices: as GCLC code, as Eukleides code, as XML rendered (by appropriate XSLT) as HTML, as XML rendered (by appropriate XSLT) in natural-language (English) form.

graphical form: with the following choices: as JPEG image, or as SVG image.

The bibliographic references are kept in BibTeX format and it is possible to get a BibTeX file with a list of selected references.

5 GeoThms Tours

In this section we will describe GeoThms framework through a series of *GeoThms Tours*, a series of paths that can be used by GeoThms users in theirs interaction with GeoThms.

5.1 Login/Registration

At the entry level (apart from the "Help" section) is the "Login" section where GeoThms users can login, or where new users can register to GeoThms. This is a standard registration form with obligatory and optional data fields and with the option to choose between a regular user, or a contributor. If a regular user wants to be a contributor, his/her request is sent to the administrator, and it is the administrator's responsibility to change the status of the user. There is also an anonymous account for a quick preliminary usage of GeoThms.

Only registered users have access to the second level. Regular users can browse the data from the database and use the drawing/proof programs in an interactive way. A contributor also has privileges to insert new data and/or update the existing ones.

5.2 Browsing

Registered users have access to the "Reports" section (see Figure 5). In this section, a user can browse through the data in the database, figures, theorems, and proofs. For each of these groups a list of available items with related details is shown (see Figure 6).

It is also possible to see the information related to the provers, the drawers, the authors of those programs and the bibliographic references.

5.3 Adding New Data

Contributors have access to the "Add/Update" section where they can add new data and/or update existing constructions, conjectures, and proofs. Constructions and conjectures are entered by users. Only proofs generated by the built-in provers

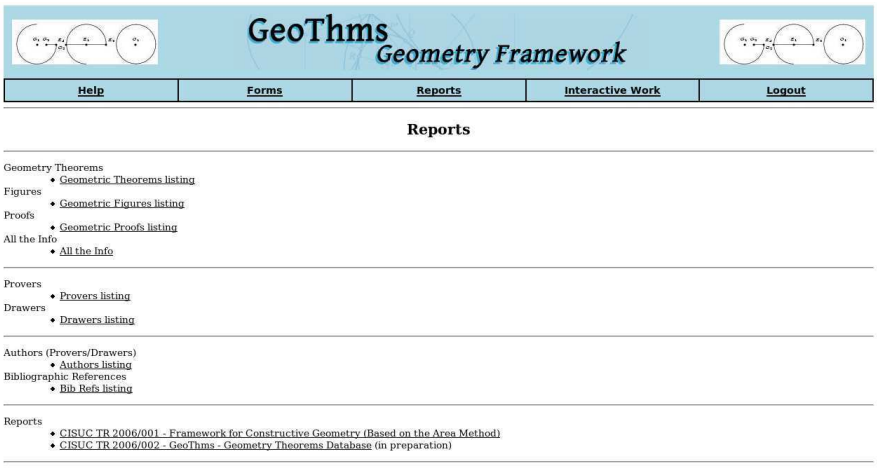


Fig. 5. GeoThms – Reports Page

can be added to the database. The relations between entities enforce that each figure and/or proof must be linked to a geometric conjecture., i.e., a contributor must first add a geometric conjecture and only after that, proceed adding a figure and/or proof to that conjecture.

A contributor can add a geometrical conjecture, its statement, corresponding figure and proof, in a single step or in separate steps. The statement must come first, and for a given conjecture more than one figure and/or proof can be added. Contributors can also update the data related to the conjectures, figures, and proofs.

It is also possible to insert/update the information about provers, drawers, authors and bibliographic references.

5.4 Interactive Work

In the "Interactive Work" section, GeoThms offers its users the possibility to use the DGSs and the ATP in a interactive way. The GeoThms user can submit the code, call for its evaluation and, if there are no errors, see the resulting figure or proof. If some syntactic, semantic or deductive error occurs an error message will

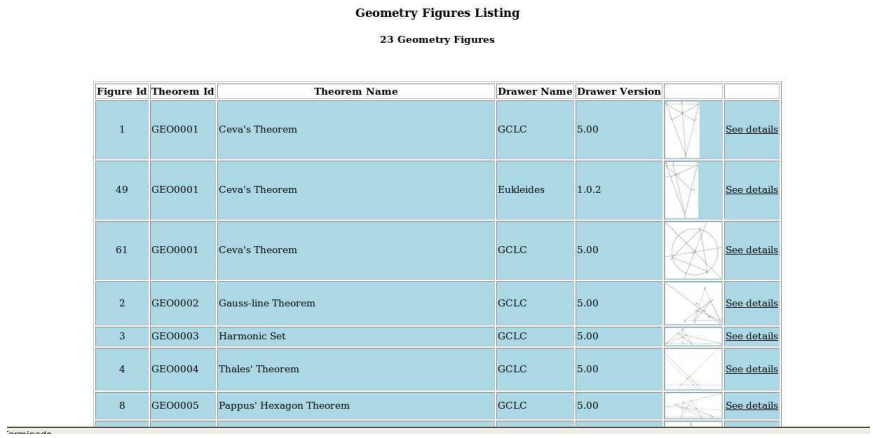
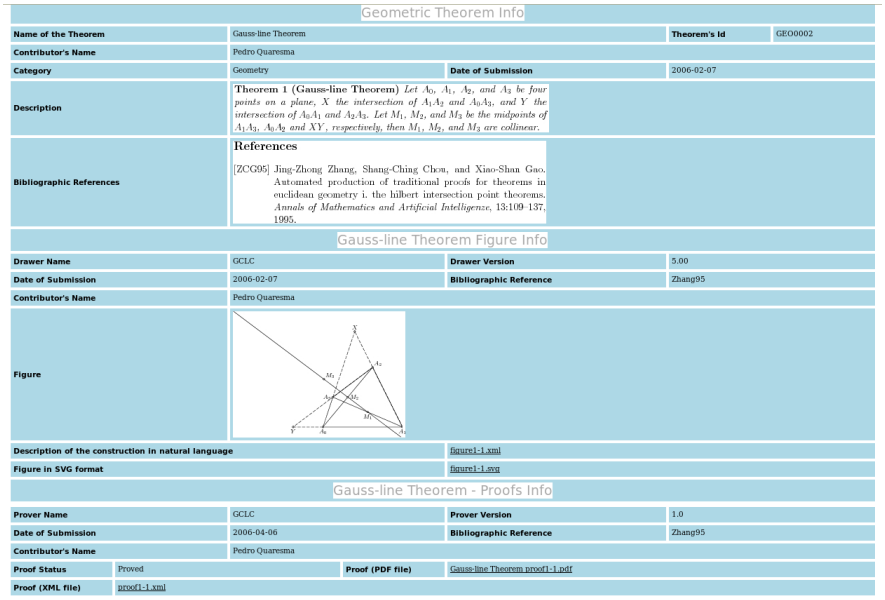


Fig. 6. GeoThms – Figures Listing



Geometric Theorem Info

Name of the Theorem	Gauss-line Theorem	Theorem's Id	GEO0002
Contributor's Name	Pedro Quaresma		
Category	Geometry	Date of Submission	2006-02-07
Description	**Theorem 1 (Gauss-line Theorem)** *Let A_0, A_1, A_2, and A_3 be four points on a plane, X the intersection of A_1A_2 and A_0A_3, and Y the intersection of A_0A_1 and A_2A_3. Let M_1, M_2, and M_3 be the midpoints of A_1A_3, A_0A_2 and XY, respectively, then M_1, M_2, and M_3 are collinear.*		
Bibliographic References	References [ZCG95] Jing-Zhong Zhang, Shang-Ching Chou, and Xiao-Shan Gao. Automated production of traditional proofs for theorems in euclidean geometry i. the hilbert intersection point theorems. *Annals of Mathematics and Artificial Intelligenze*, 13:109–137, 1995.		

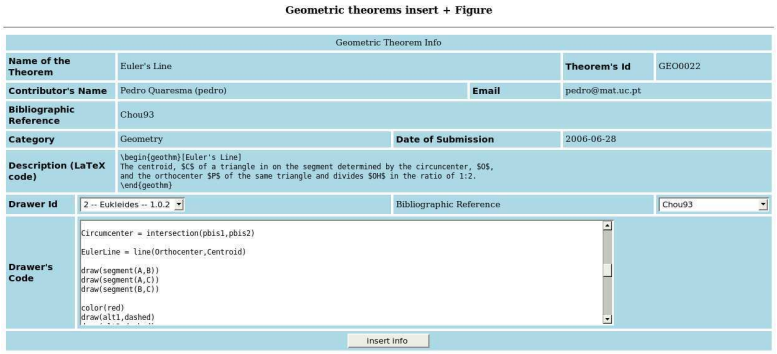
Gauss-line Theorem Figure Info

Drawer Name	GCLC	Drawer Version	5.00
Date of Submission	2006-02-07	Bibliographic Reference	Zhang95
Contributor's Name	Pedro Quaresma		
Figure			
Description of the construction in natural language		figure1-1.xml	
Figure in SVG format		figure1-1.svg	

Gauss-line Theorem - Proofs Info

Prover Name	GCLC	Prover Version	1.0
Date of Submission	2006-04-06	Bibliographic Reference	Zhang95
Contributor's Name	Pedro Quaresma		
Proof Status	Proved	Proof (PDF file)	Gauss-line Theorem proof1-1.pdf
Proof (XML file)	proof1-1.xml		

Fig. 7. GeoThms – Theorem Report



Geometric theorems insert + Figure

Geometric Theorem Info

Name of the Theorem	Euler's Line	Theorem's Id	GEO0022
Contributor's Name	Pedro Quaresma (pedro)	Email	pedro@mat.uc.pt
Bibliographic Reference	Chou93		
Category	Geometry	Date of Submission	2006-06-28
Description (LaTeX code)	\begin{geothm}[Euler's Line] The centroid, C of a triangle in on the segment determined by the circuncenter, O, and the orthocenter P of the same triangle and divides OH in the ratio of 1:2. \end{geothm}		
Drawer Id	2 -- Euklelides -- 1.0.2	Bibliographic Reference	Chou93
Drawer's Code	Circumcenter = intersection(pbis1,pbis2) EulerLine = line(Orthocenter,Centroid) draw(segment(A,B)) draw(segment(A,C)) draw(segment(B,C)) color(red) draw(alt1,dashed)		

Insert info

Fig. 8. Insertion Form

be displayed and the user is given the opportunity to correct and re-evaluate the code. Syntactic errors are errors made in description steps that are not regular with respect to syntax of the underlying geometrical language. Semantic errors occur in situation when some construction is not possible for given points (for points given by their Cartesian coordinates). Deductive errors occur in situation when some construction is always impossible (these errors require invoking the prover).

Figures 9 and 10 illustrate the possibilities to add code (within a `textarea` field), to submit it to evaluation, and to see the result graphically along with the code submitted. If there are errors, they are displayed and the user can correct them. We are planning to incorporate a syntax highlighting text editor (e.g., Helena [7]) as a substitute to the `textarea` field, providing in this way line numbering and syntax highlight.

The "Interactive Work" section can be used to work on a new result before adding it to the database. In the following text, we will illustrate an interaction of

[7] http://helene.muze.nl/

a user with GeoThms in adding, for example, the *Midpoint Theorem* to the database.

Theorem 1 (Midpoint Theorem) *Let ABC be a triangle, and let A' and B' be the midpoints of AC and BC respectively. Then the line $A'B'$ is parallel to the line AB.*

Using the interactive part of GeoThms, a user can begin by describing the construction, proceed attempting to prove the conjecture and, if all went as expected, insert all this data, along with the new conjecture statement to the database.

Describing the Construction

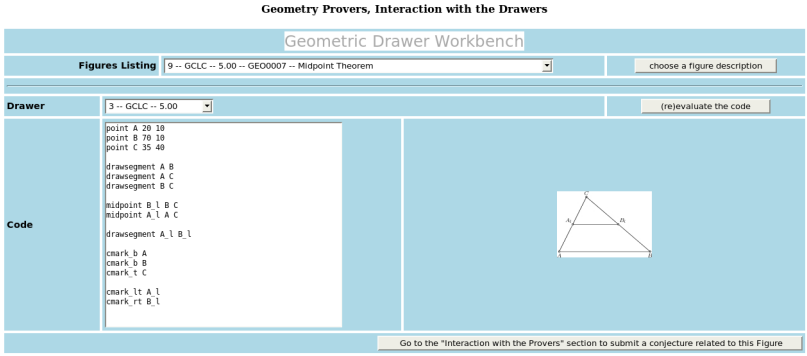


Fig. 9. Midpoint Theorem — Interaction with the DGS

The constructive specification of the figure has to define: three (fixed) points A, B, C (the vertices of the triangle); two (constructed) points A' and B', the two midpoints of AC and BC respectively, and all the "drawing" commands. Note that drawing commands are irrelevant for the theorem prover, but are relevant for producing figures (see Figure 9). The construction was made using GCLC, but the user can also use Eukleides for describing the construction, by instructions very similar to the given ones.

Testing the Conjecture

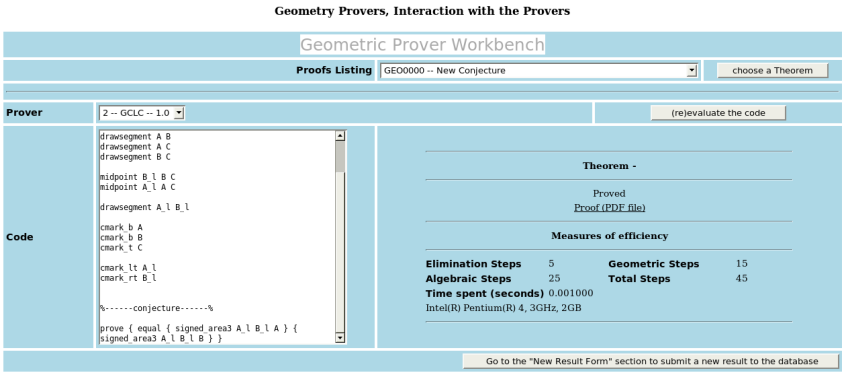


Fig. 10. Midpoint Theorem — Interaction with the ATP

After eliminating all errors from the code, the user can add a conjecture. The property to be proved can be expressed (using the area method) in the following

way $S_{A'B'A} = S_{A'B'B}$, i.e., the signed area of $\Delta A'B'A$ is equal to the signed area of $\Delta A'B'B$.

By clicking in the appropriate button, the user goes from the "Geometric Drawer Workbench" to the "Geometric Prover Workbench" with the DGS's code already in the `textarea` window. The GCLC's code can be submitted to GCLCprover without modifications, while the Eukleides' code needs to be converted with the `euktogclcprover` tool.

The user can now add the conjecture in the ATP's code:

```
prove {equal {signed_area3 A_1 B_1 A} {signed_area3 A_1 B_1 B} }
```

After that, a new cycle of writing and evaluation starts (the drawers commands are already correct, but the conjecture may be incorrectly written). After that, the user gets the output of the prover in the form of a proof status, a PDF file (generated by GCLCprover) containing the proof (if the conjecture is valid), and some measures of efficiency.

As shown in figure 10, the proof status and the measures of efficiency are accessible, and the proof is given as a PDF file. Figure 11 shows the last steps of the proof made by GCLCprover.

(11) $\left(\frac{1}{2} \cdot S_{ACB}\right) = (S_{BAB_l} + S_{BCB_l})$, by algebraic simplifications

(12) $\left(\frac{1}{2} \cdot S_{ACB}\right) = \left(\left(S_{BAB} + \left(\frac{1}{2} \cdot (S_{BAC} + (-1 \cdot S_{BAB}))\right)\right) + S_{BCB_l}\right)$, by Lemma 29 (point B_l eliminated)

(13) $\left(\frac{1}{2} \cdot S_{ACB}\right) = \left(\left(0 + \left(\frac{1}{2} \cdot (S_{ACB} + (-1 \cdot 0))\right)\right) + S_{BCB_l}\right)$, by geometric simplifications

(14) $0 = S_{BCB_l}$, by algebraic simplifications

(15) $0 = \left(S_{BCB} + \left(\frac{1}{2} \cdot (S_{BCC} + (-1 \cdot S_{BCB}))\right)\right)$, by Lemma 29 (point B_l eliminated)

(16) $0 = \left(0 + \left(\frac{1}{2} \cdot (0 + (-1 \cdot 0))\right)\right)$, by geometric simplifications

(17) $0 = 0$, by algebraic simplifications

Fig. 11. Last steps of the Proof of the Midpoint Theorem

Adding the Midpoint Theorem to the Database

With a new click, the contributor can select the "Forms" section in order to add a statement for the new construction and the corresponding figure and proof (see Figure 8).

5.5 Searching the Database

GeoThms' users can search the database over figures, conjectures, or proofs for a particular string.

6 Related work

There are several system related to GeoThms. The following ones in some degree link DGSs with ATPs or with repositories of theorems (Table 1 shows comparison between features of these tools):

Geometry Explorer combines features of DGS with a theorem prover based on the full-angle method which produces human-readable proofs (in LaTeX form) [22].

MMP/Geometer combines features of DGS and ATP, and uses different proving methods [10,11].

Geometry Expert (GEX) (new version, currently under development) is a DGS with a client-side web interface [9]; the GEX prover is based on algebraic proof methods, and the user can only select one from a limited number of conclusions (e.g., "are three selected point collinear?"). The GEX tool does not have an accessible database of problems, and does not provide a formatted output for images and proofs.

GEOTHER is an environment for manipulating and proving geometric theorems implemented in Maple, with drawing routines and the interface in Java. GEOTHER can work with a menu-driven graphic user interface and contains a collection of theorems in both elementary and differential geometry, with sample specifications that have been proved. [21,20].

Cinderella uses randomised theorem checking of the geometrical properties; it does not provide proofs in any form [5,14,19].

Discover is a DGS that can communicate with Mathematica [8], using the symbolic capabilities of the latter to implement the Gröebner bases method [2]. It is necessary to translate the geometric construction to an algebraic form and back, from the conclusion in algebraic form to its geometric counterpart. No proofs in any form are provided.

geometriagon has a vast repository of problems in the area of classical constructive (ruler and compass only) Euclidean geometry [9]. A registered user can access/edit all problems and solutions. It does not provide an ATP. The user can perform only valid steps in the construction, using only a limited set of tools, and in this way the system is capable to recognise whenever a user has reach a solution of a problem. No formal proofs are provided.

GeoView combines the Coq ATP and the GeoplanJ DGS into a framework in which it is possible to edit statements of geometrical theorems, and to visualise the statement using the DGS [1]. The proofs are not accessible.

GeoGebra is a DGS with an internationalised graphical interface allowing graphical and textual input. Figures can be exported to various formats, including a dynamic version for Web. It does not have an ATP tool, neither it keeps a repository of problems [8].

Theorema system integrates a number of different mathematical tools and reasoners, including several geometry theorem provers — provers based on Gröebner bases method, Wu's methods, the area method. The system is built on top of Mathematica [8] system and uses its visualisation tools [3].

[8] http://www.wolfram.com

[9] geometriagon: http://www.polarprof.net/geometriagon/

Tool	DGS	ATP	Readable Proofs	Web Interface	Repository of Problems	Verification of Constructions
GeoThms	√	√	√	√	√	√
Geometry Explorer	√	√	√			
MMP/Geometer	√	√	√			
GEX (old version)	√	√	√			
GEX (new version)	√	√		√		
GEOTHER	√	√			√	
Cinderella	√					
Discover	√	√				
geometriagon	√			√	√	
GeoView	√	√				
GeoGebra	√					
Theorema		√	√			

Table 1
Comparison between tools that combine DGS, ATP and repository of geometry theorems

7 Future Work

We are planning to augment the framework by other dynamic geometry tools, and other geometry theorem provers. We are considering theorem provers based on the full-area method (which also produces synthetic proofs), Wu's method and Gröebner bases method. We are planning to enable exchanging data with other tools (internally, within GeoThms, and externally) via our XML format for geometrical constructions and proofs.

We are planning to incorporate a syntax highlighting text editor as a substitute to the `textarea` field, providing in this way line numbering and syntax highlight. We are planning to use Math-ML for rendering theorems' statements and proofs stored in XML.

The search mechanism will be improved to provide options for advanced search.

We are planning to internationalise GeoThms, in order to make it wider usable in education.

The "Help" system will be improved, by adding more detailed information to the various help pages already provided.

8 Conclusions

GeoThms gives the user a complex web-based framework suitable for new ways of communicating geometric knowledge, it provides an open system where one can learn from the existing knowledge base and seek for new results. GeoThms also provides a system for storing geometric knowledge (in a strict, declarative form) — not only theorem statements, but also their (automatically generated) proofs and corresponding figures, i.e., visualisations. We are planning to further develop GeoThms by improving its functionalities and incorporating more geometrical tools. We also hope that GeoThms' growing body of geometrical constructions and formally proven geometrical theorems will become a major Internet resource for geometrical constructions and conjectures.

References

[1] Yves Bertot, Frdric Guilhot, and Locï Pottier. Visualizing geometrical statements with geoview. http://www-sop.inria.fr/lemme/geoview/geoview.ps, 2004.

[2] Francisco Botana and Jos L. Valcarce. A dynamic-symbolic interface for geometric theorem discovery. *Computers and Education*, 38:21–35, 2002.

[3] Bruno et.al. Buchberger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 2006.

[4] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. Automated production of traditional proofs for constructive geometry theorems. In Moshe Vardi, editor, *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science LICS*, pages 48–56. IEEE Computer Society Press, June 1993.

[5] Cinderella site. http://www.cinderella.de.

[6] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. Automated generation of readable proofs with geometric invariants, I. multiple and shortest proof generation. *Journal of Automated Reasoning*, 17:325–347, 1996.

[7] Mirjana Djorić and Predrag Janičić. Constructions, instructions, interactions. *Teaching Mathematics and its Applications*, 23(2):69–88, 2004.

[8] Karl Fuchs and Markus Hohenwarter. Combination of dynamic geometry, algebra and calculus in the software system geogebra. In *Computer Algebra Systems and Dynamic Geometry Systems in Mathematics Teaching Conference 2004*, Pecs, Hungary, 2004.

[9] Xiao-Shan Gao. Gex. http://www.mmrc.iss.ac.cn/~xgao/software.html.

[10] Xiao-Shan Gao and Qiang Lin. Mmp/geometer - a software package for automated geometric reasoning. In Franz Winkler, editor, *Automated Deduction in Geometry: 4th International Workshop, (ADG 2002)*, number 2930 in LNCS, pages 44–66. Springer-Verlag, 2004.

[11] X.S. Gao and Q. Lin. Mmp/geometer. http://www.mmrc.iss.ac.cn/~xgao/software.html.

[12] Predrag Janičić and Ivan Trajković. Wingclc — a workbench for formally describing figures. In *Proceedings of the 18th Spring Conference on Computer Graphics (SCCG 2003)*, pages 251–256. ACM Press, New York, USA, April, 24-26 2003.

[13] Predrag Janičić and Pedro Quaresma. System description: Gclcprover + geothms. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR 2006*, LNAI. Springer-Verlag, 2006.

[14] Ulrich Kortenkamp and Jrgen Richter-Gebert. Using automatic theorem proving to improve the usability of geometry software. In *Procedings of the Mathematical User-Interfaces Workshop 2004*, 2004.

[15] Julien Narboux. A decision procedure for geometry in coq. In *Proceedings TPHOLS 2004*, volume 3223 of *Lecture Notes in Computer Science*. Springer, 2004.

[16] Christian Obrecht. Eukleides. http://www.eukleides.org/.

[17] Pedro Quaresma and Predrag Janicic. Framework for constructive geometry (based on the area method). Technical Report 2006/001, Centre for Informatics and Systems of the University of Coimbra, 2006.

[18] Pedro Quaresma and Ana Pereira. Visualizao de construes geomtricas. *Gazeta de Matemtica*, (151), Junho 2006.

[19] Jrgen Richter-Gebert and Ulrich Kortenkamp. *The Interactive Geometry Software Cinderella*. Springer, 1999.

[20] Dongming Wang. Geother. http://www-calfor.lip6.fr/~wang/GEOTHER/.

[21] Dongming Wang. Geother 1.1: Handling and proving geometric theorems automatically. In F. Winkler, editor, *Automated Deduction in Geometry*, number 2930 in LNAI, pages 194–215, Berlin Heidelberg, 2004. Springer-Verlag.

[22] Sean Wilson and Jacques Fleuriot. Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs. In *Proceedings of UITP 2005*, 2005.

Tinycals: step by step tacticals

Claudio Sacerdoti Coen[1] Enrico Tassi[2] Stefano Zacchiroli[3]

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY

Abstract

Most of the state-of-the-art proof assistants are based on procedural proof languages, scripts, and rely on LCF tacticals as the primary tool for tactics composition. In this paper we discuss how these ingredients do not interact well with user interfaces based on the same interaction paradigm of Proof General (the *de facto* standard in this field), identifying in the coarse-grainedness of tactical evaluation the key problem. We propose *Tinycals* as an alternative to a subset of LCF tacticals, showing that the user does not experience the same problem if tacticals are evaluated in a more fine-grained manner. We present the formal operational semantics of tinycals as well as their implementation in the Matita proof assistant.

Keywords: Interactive Theorem Proving, Small Step Semantics, Tacticals

1 Introduction

Several state-of-the-art interactive theorem provers are based on procedural proof languages; the user interacts with the system mainly via a textual *script* that records the executed commands. The commands that allow progress during a proof are called *tactics* and are executed atomically. NuPRL [10], Isabelle [6], Coq [13], and Matita[4] (the proof assistant under development by our team at the University of Bologna) are a few examples of those systems.

The best known proof assistant that provides only a declarative proof language is Mizar [8], while a few others superpose a declarative proof language on top of a procedural core. The most notable system in this category is Isabelle, which in its Isabelle/Isar variant offers to users the declarative Isar proof language [14].

With the exception of Mizar, both kind of systems share the same user interface paradigm, inspired by the pioneering work on CtCoq [2] and now incarnated by Proof General [1]. In this paradigm, the smallest amount of code that can be

[1] sacerdot@cs.unibo.it
[2] tassi@cs.unibo.it
[3] zacchiro@cs.unibo.it
[4] http://matita.cs.unibo.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

```
theorem lt_O_defactorize_aux:
 \forall f:nat_fact.
 \forall i:nat.
 O < defactorize_aux f i.
intro. elim f.
simplify. unfold lt.
rewrite > times_n_SO.
apply le_times.
change with (O < \pi _ i).
apply lt_O_nth_prime_n.
change with (O < (\pi _ i)^n).
apply lt_O_exp.
apply lt_O_nth_prime_n.
simplify.unfold lt.
rewrite > times_n_SO.
apply le_times.
change with (O < (\pi _ i)^n).
apply lt_O_exp.
apply lt_O_nth_prime_n.
change with
  (O < defact n1 (S i)).
apply H.
```

```
theorem lt_O_defactorize_aux:
 \forall f:nat_fact.
 \forall i:nat.
 O < defactorize_aux f i.
intro; elim f;
[1,2:
  simplify; unfold lt;
  rewrite > times_n_SO;
  apply le_times;
  [ change with (O < \pi _ i);
    apply lt_O_nth_prime_n
  |2,3:
    change with (O < (\pi _ i)^n);
    apply lt_O_exp;
    apply lt_O_nth_prime_n
  | change with
      (O < defact n1 (S i));
    apply H ] ].
```

Fig. 1. The same proof with (on the right) and without (on the left) tacticals.

executed atomically is the *statement*, which during a proof is either a tactic (in the procedural world) or a single proof step (in the declarative world).

Scripts can be understood only by step by step execution, getting feedback on the proof status from the system. Since feedback is given only between atomic steps (at the so called *execution points*), it is important to have atomic steps as small as possible for the sake of understanding but, also of debugging and proof maintenance. This is in contrast with *tacticals*, higher order constructs which can be used to combine tactics together.

In this paper we propose a replacement for tacticals in order to obtain smaller atomic execution steps. Our work is not relevant in the context of declarative proof languages. However, those few systems where it is possible to embed procedural scripts inside declarative proof steps may already provide the functionality we suggest.

Tacticals first appeared in the LCF theorem prover [5] in 1979. Paradigmatic examples of tacticals are *sequential composition* and *branching*.[5] The former, usually written as "t_1 ; t_2", takes two tactics t_1 and t_2 and apply t_2 to each of the conjectures resulting from the application of t_1 to the current conjecture (of course its application can be repeated to obtain *pipelines* of tactics "t_1 ; t_2 ; t_3 ;$\cdots$"). The latter, "t ;[t_1 | $\cdots$ | t_n]", takes $n+1$ tactics, applies t to the current conjecture and, requiring t to return exactly n conjectures, applies t_1 to the first returned conjecture, t_2 to the second, and so forth.

Tacticals improve procedural proof languages providing concrete advantages, that we illustrate with Figure 1. The concrete syntax used in the figure is that of the Matita proof assistant.

Proof structuring. Using branching, the script representation of proofs can mimic the structure of the proof tree (the tree having conjectures as nodes and tactic-labeled arcs). Since proof tree branches usually reflect conceptual parts of the pen and paper proof, the branching tactical helps in improving scripts readability (on the average very poor, if compared with declarative proof languages). Even

[5] In this paper the term "branching" is used to refer to LCF's `THENL` tactical

maintainability of proof scripts is improved by the use of branching, for example when hypothesis are added, removed or permuted.

For instance, in the right hand side of Figure 1 it is now clear that `elim f` splits the proof in two branches; both of them (selected by "`[1,2:`") begin with the same tactics until each branch is split again by the application of the `le_times` lemma. Of the four branches, the second and third one (selected by "`|2,3:`") are proved by the same tactics, being proofs of the same fact. All the tactics that are not followed by branching do not introduce ramifications in the proof.

In practice, the proof on the left hand side would be written by using indentation and blank lines to understand where branches start and end. This way readability is improved, but a lesser effect is achieved for proof maintenance. Moreover, the system does not verify in any way the layout of the proof and does not guarantee consistency when the script is changed. We expect that users will abandon this behaviour as soon as an alternative without drawbacks — not the case of LCF tacticals — will surface.

Notice that the selection of multiple branches at a time we propose in this paper is an improvement over the standard branching tactical.

Conciseness. As code factorization is a good practice in programming, proof factorization is in theorem proving. The use of tacticals like sequential composition reduce the need of copy-and-paste in proof scripts helping in factorizing common cases in proofs (so frequent in formal proofs pertaining to the computer science field). Conciseness is evident in Figure 1.

In all the proof assistants we are aware of, tacticals are evaluated atomically and placing the execution point in the middle of complex tacticals (for example at occurrences of "**;**" in tactic pipelines) is not allowed. In Figure 1, this means that having the execution point at the beginning of the proof and asking the system to move it forward (i.e. to execute the next statement), the user will result in a "proof completed" status, without having any feedback of the inner proof status the system passed through. The only way for the user to inspect those status — a frequent need, for instance for script maintenance or proof presentation — is to manually de-structure the complex tacticals.

The big step evaluation of tacticals has also drawbacks on how proof authors develop their proofs. Since it is not always possible to predict the outcome of complex tactics, the following is common practice:

(i) evaluate the next tactic of the script;

(ii) inspect the set of returned conjectures;

(iii) decide whether the use of "**;**" or "[" is appropriate;

(iv) if it is: retract the last statement, add the tactical, go to step (i).

Last, but not less important, is the imprecise error reporting of big step evaluation of tacticals. Consider the frequent case of a script breaking and the user having to fix it. The error message returned by the system may concern an inner status unknown to the user, since the whole tactical is evaluated at once. Moreover, the error message will probably concern terms that do not appear verbatim in the script. Finding the statement that need to be fixed is usually done replacing tactics

with identity tactic proceeding outside-in, until the single failing tactic is found. This technique is not only error prone, but is even not reliable in presence of *side-effects* (tactics closing conjectures other than that on which they are applied), since the identity tactic has no side-effects and branches of the proof may be affected by their absence.

We claim that the tension between tacticals and Proof General like interfaces can be broken. In this paper we present a tiny language of tacticals — the so called *tinycals* — which solves this issue. Tinycals can be evaluated in small steps, enabling the execution point to be placed inside complex structures like pipelines or branching constructs. This goal is achieved by de-structuring the syntax of tacticals and stating the semantics as a transition system over *evaluation status*, that are structures richer than the proof status tactics act on. Note that de-structuring does not necessarily mean changing the concrete syntax of tacticals, but rather enabling parsing and immediate evaluation of tactical fragments like "[" alone.

The paper is organized as follows. Section 2 describes the abstract syntax of tinycals together with their small-step operational semantics. Other advantages of tinycals with respect to LCF tacticals are discussed there as well. Section 3 presents the tinycals implementation in Matita. Section 4 deals with tacticals not covered by tinycals. Section 5 discusses related work and Section 6 concludes the paper.

2 Tinycals: syntax and semantics

The grammar of tinycals is reported in Table 1, where $\langle L \rangle$ is the top-level nonterminal generating the script language. $\langle L \rangle$ is a sequence of statements $\langle S \rangle$. Each statement is either an atomic tactical $\langle B \rangle$ (marked with "`tactic`") or a tinycal.

Note that the part of the grammar related to the tinycals themselves is completely de-structured. The need for embedding the structured syntax of LCF tacticals (nonterminal $\langle B \rangle$) in the syntax of tinycals will be discussed in Section 4. For the time being, the reader can suppose the syntax to be restricted to the case $\langle B \rangle ::= \langle T \rangle$.

We will now describe the semantics of tinycals which is parametric in the proof status tactics act on and also in their semantics (see Table 2).

A *proof status* is the logical status of the current proof. It can be seen as the current proof tree, but there is no need for it to actually be a tree. Matita for instance just keeps the set of conjectures to prove, together with a proof term where meta-variables occur in place of missing components. From a semantic point of view the proof status is an abstract data type. Intuitively, it must describe at least the set of conjectures yet to be proved. A *Goal* is another abstract data type used to index conjectures.

The function *apply_tac* implements tactic application. It consumes as input a tactic, a proof status, and a goal (the conjecture the tactic should act on), and returns as output a proof status and two lists of goals: the set of newly opened goals and the set of goals which have been closed. This choice enables our semantics to account for *side-effects*, that is: tactics can close goals other than that on which they have been applied, a feature implemented in several proof assistants via existential or meta-variables [4,9]. The proof status was not directly manipulated by tactics in

Table 1
Abstract syntax of tinycals and core LCF tacticals.

$\langle S\rangle$ ::=		(**statements**)	$\langle L\rangle$ ::=		(**language**)
	“`tactic`” $\langle B\rangle$	(tactic)		$\langle S\rangle$	(statement)
|	“**.**”	(dot)	|	$\langle S\rangle\ \langle S\rangle$	(sequence)
|	“**;**”	(semicolon)	$\langle B\rangle$ ::=		(**tacticals**)
|	“[”	(branch)		$\langle T\rangle$	(tactic)
|	“|”	(shift)	|	“`try`” $\langle B\rangle$	(recovery)
|	$i_1,\ldots,i_n$ “**:**”	(projection)	|	“`repeat`” $\langle B\rangle$	(looping)
|	“$*$ **:**”	(wild card)	|	$\langle B\rangle$ “**;**” $\langle B\rangle$	(composition)
|	“`accept`”	(acknowledge)	|	$\langle B\rangle$ “**;[**” $\langle B\rangle$ “|” ... “|” $\langle B\rangle$ “]”	(branching)
|	“]”	(merge)			
|	“`focus`” $[g_1;\cdots;g_n]$	(selection)	$\langle T\rangle$ ::= ...		(**tactics**)
|	“`done`”	(de-selection)			

Table 2
Semantics parameters.

proof status:	ξ
proof goal:	*goal*
tactic application:	$apply\_tac : T \to \xi \to goal \to \xi \times goal\ \texttt{list} \times goal\ \texttt{list}$

LCF because of the lack of meta-variables and side effects.

In the rest of this section we will define the semantics of tinycals as a transition (denoted by $\longrightarrow$) on evaluation status. *Evaluation status* are defined in Table 3.

The first component of the status (*code*) is a list of statements of the tinycals grammar. The list is consumed, one statement at a time, by each transition. This choice has been guided by the un-structured form of our grammar and is the heart of the fine-grained execution of tinycals.

The second component is the proof status, which we enrich with a *context stack* (the third component). The context stack, a representation of the proof history so far, is handled as a stack: levels get pushed on top of it either when the branching tinycal “[” is evaluated, or when “`focus`” is; levels get popped out of it when the corresponding closing tinycals are (“]” for “[” and “`done`” for “`focus`”). Since the syntax is un-structured, we can not ensure statically proper nesting of tinycals, therefore each stack level is equipped with a *tag* which annotates it with the creating tinycal (`B` for “[” and `F` for “`focus`”). In addition to the tag, each stack level has three components Γ, τ and κ respectively for active tasks, tasks postponed to the

Table 3
Evaluation status.

$task$	$=$ `int` $\times$ (`Open` $goal$ $\mid$ `Closed` $goal$)	(task)
Γ	$=$ $task$ `list`	(context)
τ	$=$ $task$ `list`	("todo" list)
κ	$=$ $task$ `list`	(dot's continuations)
$ctxt\_tag$	$=$ `B` $\mid$ `F`	(stack level tag)
$ctxt\_stack$	$=$ $(\Gamma \times \tau \times \kappa \times ctxt\_tag)$ `list`	(context stack)
$code$	$=$ $\langle S \rangle$ `list`	(statements)
$status$	$=$ $code \times \xi \times ctxt\_stack$	(evaluation status)

end of branching and tasks posponed by "•". The role of these componenets will be explained in the description of the tinycals that acts on them. Each component is a sequence of numbered tasks. A *task* is an handler to either a conjecture yet to be proved, or one which has been closed by a side-effect. In the latter case the user will have to confirm the instantiation with "`accept`".

Each evaluation status is meaningful to the user and can be presented by slightly modifying already existent user interfaces. Our presentation choice is described in Section 3. The impatient reader can take a sneak preview of Figure 2, where the interesting part of the proof status is presented as a notebook of conjectures to prove, and the conjecture labels represent the relevant information from the context stack by means of: 1) bold text (for conjectures in the currently selected branches, targets of the next tactic application; they are kept in the Γ component of the top of the stack); 2) subscripts (for not yet selected conjectures in sibling branches; they are kept in the Γ component of the level below the top of the stack). The rest of the information hold in the stack does not need to be shown to the user since it does not affect immediate user actions.

We describe first the semantics of the tinycals that do not involve the creation of new levels on the stack. The semantics is shown in Table 4, where some utility functions (described in Appendix A) are used.

Tactic application

Consider the first case of the tinycals semantics of Table 4. It makes use of the first component (denoted Γ) of a stack level, which represent the "current" goals, that is the set of goals to which the next tactic evaluated will be applied.

When a tactic is evaluated, the set Γ of current goals is inspected (expecting to find at least one of them), and the tactic is applied in turn to each of them in order to obtain the final proof status. At each step i the two sets C_i^o and G_i^c of goals opened and closed so far are updated. This process is atomic to the user (i.e. no feedback is given while the tactic is being applied to each of the current goals in turn), but she is free to cast off atomicity using branching. After the tactic has been applied to all goals, the new set of current goals is created containing all the goals which have been opened during the applications, but not already closed.

Table 4
Basic tinycals semantics.

$$\langle \text{“}\texttt{tactic}\text{”}\ \langle T\rangle :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi_n, S'\rangle \qquad n \geq 1$$

where $[g_1;\cdots;g_n] = \mathit{get\_open\_goals\_in\_tasks\_list}(\Gamma)$

and
$$\begin{cases} \langle \xi_0, G^o_0, G^c_0\rangle = \langle \xi, [\,], [\,]\rangle & \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1}\rangle = \langle \xi_i, G^o_i, G^c_i\rangle & g_{i+1} \in G^c_i \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1}\rangle = \langle \xi', (G^o_i \setminus G^c) \cup G^o, G^c_i \cup G^c\rangle & g_{i+1} \notin G^c_i \\ \quad \text{where } \langle \xi', G^o, G^c\rangle = \mathit{apply\_tac}(T, \xi_i, g_{i+1}) & \end{cases}$$

and $S' = \langle \Gamma', \tau', \kappa', t\rangle :: \mathit{close\_tasks}(G^c_n, S)$

and $\Gamma' = \mathit{mark\_as\_handled}(G^o_n)$

and $\tau' = \mathit{remove\_tasks}(G^c_n, \tau)$

and $\kappa' = \mathit{remove\_tasks}(G^c_n, \kappa)$

$$\langle \text{“}\texttt{;}\text{”} :: c, \xi, S\rangle \longrightarrow \langle c, \xi, S\rangle$$

$$\langle \text{“}\texttt{accept}\text{”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $\Gamma = [\langle j_1, \texttt{Closed}\ g_1\rangle; \cdots ; \langle j_n, \texttt{Closed}\ g_n\rangle] \qquad n \geq 1$

and $G^c = [g_1;\cdots;g_n]$

and $S' = \langle [\,], \mathit{remove\_tasks}(G^c, \tau), \mathit{remove\_tasks}(G^c, \kappa), t\rangle$
$:: \mathit{close\_tasks}(G^c, S)$

$$\langle \text{“}\texttt{.}\text{”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, \langle [l_1], \tau, [l_2;\cdots;l_n] \cup \kappa, t\rangle :: S\rangle \qquad n \geq 1$$

where $\mathit{get\_open\_tasks}(\Gamma) = [l_1;\cdots;l_n]$

$$\langle \text{“}\texttt{.}\text{”} :: c, \xi, \langle \Gamma, \tau, l :: \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, \langle [l], \tau, \kappa, t\rangle :: S\rangle$$

where $\mathit{get\_open\_tasks}(\Gamma) = [\,]$

They are marked (using the *mark_as_handled* utility) so that they do not satisfy the *unhandled* predicate, indicating that some tactic has been applied to them. Goals closed by side effects are removed from τ and κ and marked as `Closed` in S. The reader can find a datailed description of this procedure in Appendix A.

Sequential composition

Since sequencing is handled by Γ, the semantics of "**;**" is simply the identity function. We kept it in the syntax of tinycal for preserving the parallelism with LCF tacticals.

Side-effects handling

"`accept`" (third case in Table 4) is a tinycal used to deal with side-effects. Consider for instance the case in which there are two current goals on which the user branches. It can happen that applying a tactic to the first one closes the second, removing the need of the second branch in the script. Using tinycals the user will never see branches she was aware of disappear without notice. Cases like the above one are thus handled marking the branch as `Closed` (using the *close_tasks* utility) on the stack and requiring the user to manually acknowledge what happened on it using the "`accept`" tinycal, preserving the correspondence among script structure and proof tree.

Example 2.1 Consider the following script:

```
apply trans_eq; [ apply H | apply H1 | accept ]
```

where the application of the transitivity property of equality to the conjecture $L = R$ opens the three conjectures $?_1 : L =?_3$, $?_2 :?_3 = R$ and $?_3 : nat$. Applying the hypothesis H instantiates $?_3$, implicitly closing the third conjecture, that thus has to be acknowledged.

Local de-structuring

Structuring proof scripts enhances their readability as long as the script structure mimics the structure of the intuition behind the proof. For this reason, authors do not always desire to structure proof scripts down to the most far leaf of the proof tree.

Example 2.2 Consider for instance the following script snippet template:

```
tac1;
[ tac2. tac3.
| tac4; [ tac5 | tac6 ] ]
```

Here the author is trying to mock-up the structure of the proof (two main branches, with two more branches in the second one), without caring about the structure of the first branch.

Tacticals do not allow un-structured scripts to be nested inside branches. In the example, they would only allow to replace the first branch with the identity tactic, continuing the un-structured snippet "tac2. tac3." at the end of the snippet, but this way the correspondence among script structure and proof tree would be completely lost. The semantics of the tinycal "**.**" (last two cases of Table 4) accounts for local use of un-structured script snippets.

When "**.**" is applied to a non-empty set of current goals, the first one is selected and become the new singleton current goals set Γ. The remaining goals are remembered in the third component of the current stack level (*dot's continuations*,

denoted κ), so that when the "." is applied again on an empty set of goals they can be recalled in turn. The locality of "." is inherited by the locality of dot's continuation κ to stack levels.

Table 5
Branching tinycals semantics.

$$\langle \text{“[”} :: c, \xi, \langle [l_1; \cdots; l_n], \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle \qquad n \geq 2$$

where $\mathit{renumber\_branches}([l_1; \cdots; l_n]) = [l'_1; \cdots; l'_n]$

and $S' = \langle [l'_1], [\,], [\,], \mathtt{B}\rangle :: \langle [l'_2; \cdots; l'_n], \tau, \kappa, t\rangle :: S$

$$\langle \text{“|”} :: c, \xi, \langle \Gamma, \tau, \kappa, \mathtt{B}\rangle :: \langle [l_1; \cdots; l_n], \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle \qquad n \geq 1$$

where $S' = \langle [l_1], \tau \cup \mathit{get\_open\_tasks}(\Gamma) \cup \kappa, [\,], \mathtt{B}\rangle :: \langle [l_2; \cdots; l_n], \tau', \kappa', t'\rangle :: S$

$$\langle i_1, \ldots, i_n \text{“:”} :: c, \xi, \langle [l], \tau, [\,], \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $\mathit{unhandled}(l)$

and $\forall j = 1 \ldots n, \quad \exists l_j = \langle j, s_j\rangle, \quad l_j \in l :: \Gamma'$

and $S' = \langle [l_1; \cdots; l_n], \tau, [\,], \mathtt{B}\rangle :: \langle (l :: \Gamma') \setminus [l_1; \cdots; l_n], \tau', \kappa', t'\rangle :: S$

$$\langle \text{“}*\text{:”} :: c, \xi, \langle [l], \tau, [\,], \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $\mathit{unhandled}(l)$

and $S' = \langle l :: \Gamma', \tau, [\,], \mathtt{B}\rangle :: \langle [\,], \tau' \cup \mathit{get\_open\_tasks}(\Gamma) \cup \kappa, \kappa', t'\rangle :: S$

$$\langle \text{“]”} :: c, \xi, \langle \Gamma, \tau, \kappa, \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $S' = \langle \tau \cup \mathit{get\_open\_tasks}(\Gamma) \cup \Gamma' \cup \kappa, \tau', \kappa', t'\rangle :: S$

$$\langle \text{“}\mathtt{focus}\text{”} \ [g_1; \cdots; g_n] :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $g_i \in \mathit{get\_open\_goals\_in\_status}(S)$

and $S' = \langle \mathit{mark\_as\_handled}([g_1; \cdots; g_n]), [\,], [\,], \mathtt{F}\rangle$

$$:: \mathit{close\_tasks}(\langle \Gamma, \tau, \kappa, t\rangle :: S)$$

$$\langle \text{“}\mathtt{done}\text{”} :: c, \xi, \langle [\,], [\,], [\,], \mathtt{F}\rangle :: S\rangle \longrightarrow \langle c, \xi, S\rangle$$

Table 5 describes the semantics of tinycals that require a stack discipline.

Branching

Support for branching is implemented by "[", which creates a new level on the stack for the first of the current goals. Remaining goals (the current *branching context*) are stored in the level just below the freshly created one. There are three different ways of selecting them. Repeated uses of "|" consume the branching context in sequential order. $i_1,\ldots,i_n$"**:**" enables multiple positional selection of goals from the branching context. "$*$**:**" recall all goals of the current branching context as the new set of current goals. The semantics of all these branching tacticals is shown in the first five cases of Table 5.

Each time the user finishes working on the current goals and selects a new goal from the branching context, the result of her work (namely the current goals in Γ) needs to be saved for restoring at the end of the branching construct. This is needed to implement the LCF semantics that provides support for snippets like the following:

Example 2.3

```
tac1; [ tac2 | tac3 ]; tac4
```

where the goals resulting by the application of `tac2` *and* `tac3` are re-flowed together to create the goals set for `tac4`.

The place where we store them is the second component of stack levels (*todo list*, denoted τ). Each time a branching selection tinycal is used the current goals set (possibly empty) is appended to the todo list for the current stack level.

When "]" is used to finish branching (fifth rule of Table 5), the todo list τ is used to create the new set of current goals Γ, together with the goals not handled during the branching (note that this is a small improvement over LCF tactical semantics, where leaving not handled branches is not allowed).

Focusing

The pair of tinycals "`focus`"... "`done`" is similar in spirit to the pair "["... "]", but is not required to work on the current branching context. With "`focus`", goals located everywhere on the stack can be recalled to form a new set of current goals. On this the user is then free to work as she prefer, for instance branching, but is required to close all of them before invoking "`done`".

The intended use of "`focus`"... "`done`" is to deal with meta-variables and side effects. The application of a tactic to a conjecture with meta-variables in the conclusion or hypotheses can instantiate the meta-variables making other conjectures false. In other words, in presence of meta-variables conjectures are no longer independent and it becomes crucial to consider and close a bunch or dependent conjectures together, even if in far away branches of the proof. In these cases "`focus`"... "`done`" is used to select all the related branches for immediate work on them. Alternatively, "`focus`"... "`done`" can be used to jump on a remote branch of the tree in order to instantiate a meta-variable by side effects before resuming proof search from the current position.

Note that using "`focus`"... "`done`", no harm is done to the proper structuring of scripts, since all goals the user is aware of, if closed, will be marked as `Closed`

requiring her to manually "`accept`" them later on in the proof.

3 Implementation issues

Tinycals have been implemented in the Matita proof assistant. This section describes the issues faced in their implementation.

Encoding of tacticals

Tacticals play two different roles in a proof assistant. They can be used both in scripts and in tactic implementations. As a matter of fact at least one tactical among sequential composition and branching is used in the implementation of each derived tactic.

In this paper we propose the replacement of tacticals with tinycals. Tacticals operate on proof status, while tinycals operate on evaluation status. This is welcome when tinycals are used in scripts, since the additional information kept in the evaluation status is the rich intermediate state we want to present to the user. On the contrary, this datatype change does not allow the replacement of tacticals with tinycals in the implementation of derived tactics. Thus we are immediately led to consider if it is possible to express tacticals in terms of tinycals, in order to avoid an independent re-implementation of related operations.

The answer is positive under additional assumptions on the abstract data type of proof status. Intuitively, we need to define two "inverse" functions to embed a proof status, a goal, and a code in an evaluation status (let it be *embed*) and to project an evaluation status to a proof status and two lists of opened and closed goals (let it be *proj*). Once the two functions are implemented, we can express sequential composition and branching as follows:

$$(t_1; t_2)(\xi, g) = \mathit{proj}(\mathit{eval}(\mathit{embed}([t_1; \text{“;”}; t_2], \xi, g))) \quad (1)$$

$$(t; [t_1 | \ldots | t_n])(\xi, g) = \mathit{proj}(\mathit{eval}(\mathit{embed}([t; \text{“[”}; t_1; \text{“|”}; \ldots; \text{“|”}; t_n; \text{“]”}], \xi, g))) \quad (2)$$

where *eval* is the transitive closure of $\longrightarrow$. For each status S the code of the status $\mathit{eval}(S)$ is empty.

The *embed* function is easily defined as:

$$\mathit{embed}(c, \xi, g) = \langle c, \xi, [\langle g, [\,], [\,], \mathtt{F}\rangle]\rangle$$

To define the *proj* function, however, we need to be able to compute the set of goals opened and closed by $\mathit{eval}(\mathit{embed}(c, \xi, g))$ for any given code c, proof status ξ and selected goal g. The formers are easily computed by the *get_open_goals_in_status* utility of Appendix A. However, to compute the latter the information stored in an evaluation context is not enough.

We say that tactics *do not reuse goals* whenever closed goals cannot be re-opened (remember that a goal is just an handle to a conjecture, not the conjecture itself). Concretely, it is possible to respect this property in the implementation by keeping a global counter that represents the highest goal index already used. When a tactic opens a new goal it picks the successor of the counter, that is also incremented. When tactics do not reuse goals it is possible to determine the goals closed by a

sequence of evaluation steps by comparing the set of open goals at the two extremes of the sequence. To make this comparison it is possible to add to the proof status abstract data type a method that returns the set of opened goals.

Let *diff* be the function that given two proof status ξ and ξ' returns the set of goals that were open in ξ and are closed in ξ'. For each proof status ξ the $proj_\xi$ function is defined as:

$$proj_\xi([\,], \xi', S) = (\xi', get\_open\_goals\_in\_status(S), diff\,(\xi, \xi'))$$

The function $proj_\xi$ must be used in Equation (1) and Equation (2) in place of *proj*.

Tinycals user interface

Tinycals would be worthless without a way to present evaluation status to the user. Our current solution for the Matita user interface is shown in Figure 2.

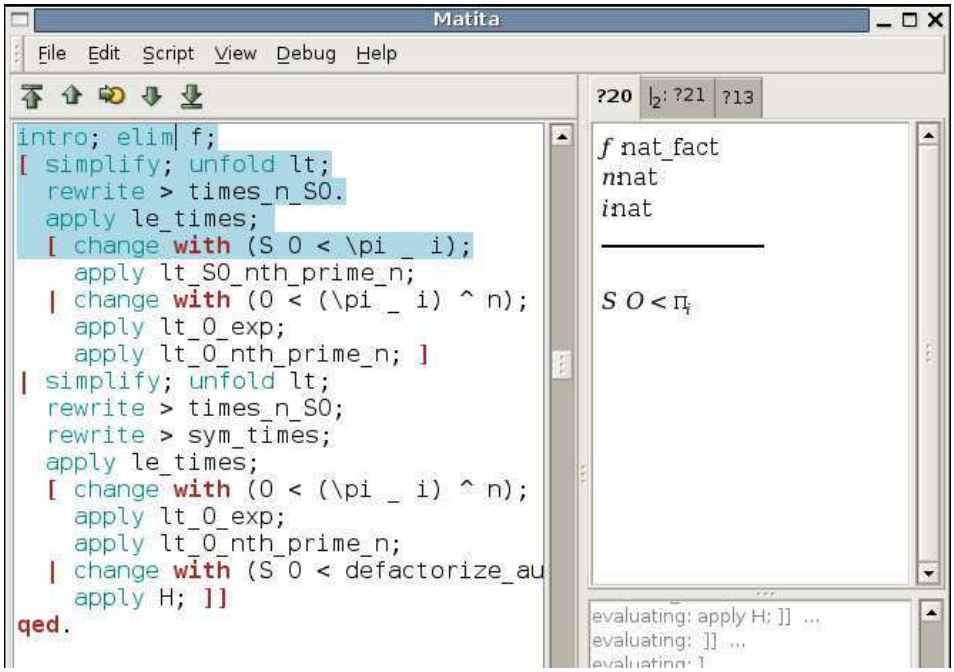


Fig. 2. Evaluation status representation in the Matita user interface.

We already had a Proof General like user interface with script and execution point (on the left of Figure 2) and a tabbed representation of the set of open conjectures (on the right) as sequents, using meta-variable indexes as labels. What the user was missing to work with tinycals was a visual representation of the stack. Our choice has been to represent the current branching context as tab label annotations: all goals in the current goals set have their labels typeset in boldface, goals of the current branching context have labels prepended by $|_n$ (where n is their positional index), and goals already closed by side-effects have strike-through labels like: ~~$?_n$~~.

For instance in Figure 2, the only goal (in bold-face) the next tactic will be applied to is 20 (i.e. $\Gamma = [\langle 1, \texttt{Open}\ 20\rangle]$), while goal 21 will be selected by the next "|" tinycal.

This choice makes the user aware of which goals will be affected by a tactic evaluated at the execution point, and of all the indexing information she might need there. She indeed can see all meta-variable indexes (in case she wants to "`focus`") and all the positional indexes of goals in the current branching context (for $i_1,\ldots,i_n$ "`:`" and "`*:`"). Yet, this user interface choice minimizes the drift from the usual way of working with Proof General like interfaces.

4 A digression on the remaining tacticals

Of the basic LCF tacticals, we have considered so far only sequential composition and branching. It is worth discussing the remaining ones, in particular *try*, || (or-else) and *repeat*.

The *try* T tactical, that never fails, applies the tactic T, behaving as the identity if T fails. It is a particular case of the or-else tactical: $T_1||T_2$ behaves as T_1 if T_1 does not fail, as T_2 otherwise. Thus *try* T is equivalent to $T||id$.

The try and or-else tacticals occur in a script with two different usages. The most common one is after sequential composition: $T_1; try\ T_2$ or $T_1; T_2||T_3$. Here the idea is that the user knows that T_2 can be applied to some of the goals generated by T_1 (and T_3 to the others in the second case). So she is faced with two possibilities: either use branching and repeat T_2 (or T_3) in every branch, or use sequential composition and backtracking (encapsulated in the two tacticals). Tinycals offer a better solution to either choice by means of the projection and wild card tinycals: $T_1; [i_1,\ldots,i_n : T_2|* : T_3]$. The latter expression is not also more informative to the reader, but it is also computationally more efficient since it avoids the (maybe costly) application of T_2 to several goals.

The second usage of try and or-else is inside a repeat tactical. The *repeat* T tactical applies T once, failing if T fails; otherwise the tactical recursively applies T again on every goal opened by T until T fails, in which case it behaves as the identity tactic.

Is it possible to provide an un-structured version of *try* T, $T||T'$, and *repeat* T in the spirit of tinycals in order to allow the user to write and execute T step by step inspecting the intermediate evaluation status? The answer is negative as we can easily see in the simplest case, that of *try* T. Consider the statement $T; try\ (T_1; T_2)$ where sequential composition is supposed to be provided by the corresponding tinycal. Let T open two goals and suppose that "*try*" is executed atomically so that the evaluation point is just before T_1. When the user executes T_1, T_1 can be applied as expected to both goals in sequence. Let ξ be the proof status after the application of T and let ξ_1 and ξ_2 be those after the application of T_1 to the first and second goal respectively. Let now the user execute the identity tinycal ";" followed by T_2 and let T_2 fail over the first goal. To respect the intended semantics of the tactical, the status ξ_2 should be *partially* backtracked to undo the changes from ξ to ξ_1, preserving those from ξ_1 to ξ_2.

If the system has side effects the latter operation is undefined, since T_1 applied to ξ could have instantiated meta-variables that controlled the behavior of T_1 applied to ξ_1. Thus undoing the application of T_1 to the first goal also invalidates the previous application of T_1 to the second goal.

Even if the system has no side effects, the requirement that proof status can be partially backtracked is quite restrictive on the possible implementations of a proof status. For instance, a proof status cannot be a simple proof term with occurrences of meta-variables in place of conjectures, since backtracking a tactic would require the replacement of a precise subterm with a meta-variable, but there would be no information to detect which subterm.

As a final remark, the simplest solution of implementing partial backtracking by means of a full backtrack to ξ followed by an application of T_1 to the second goal only does not conform to the spirit of tinycals. With this implementation, the application of T_1 to the second goal would be performed twice, sweeping the waste of computational resources under the rug. The only honest solution consists of keeping all tacticals, except branching and sequential composition, fully structured as they are now. The user that wants to inspect the behavior of $T; try\ T_1$ before that of $T; try\ (T_1; T_2)$ is obliged to do so by executing atomically $try\ T_1$, backtracking by hand and executing $try\ (T_1; T_2)$ from scratch. A similar conclusion is reached for the remaining tacticals. For this reason in the syntax given in Table 1 the production $\langle B \rangle$ lists all the traditional tacticals that are not subsumed by tinycals. Notice that atomic sequential composition and atomic branching (as implemented in the previous section) are also listed since tinycals cannot occur as arguments of a tactical.

5 Related work

Different presentations of the semantics of tacticals has been given in the past. The first presentation has been given in [5] by Gordon et al. Although a larger set of tacticals than that considered here was described in their work, the problem of inspection of inner proof status was not considered. Proof General-like interfaces were not available at the time, as well as meta-variables and tactics with side-effects.

In [7], Kirchner described a small step semantics of Coq tacticals. Despite the minor expressive advantages offered by tinycals over the corresponding Coq tacticals (like "`focus`", "$*$**:**", $i_1,\ldots,i_n$"**:**", the less constrained use of "[", and the structuring facilities implemented by "**.**" and "`accept`"), the formalization of tinycals is more general and we believe that it can be applied to a large class of proof assistants. In particular our semantics only assume an abstract proof status and a very general type for tactic applications, while in [7] a very detailed API for proof trees was assumed.

Delahaye in [3] described $\mathcal{L}_{tac}$, a powerful meta-language which can be used both by users and tactics implementors to write small automations at the proof language level. $\mathcal{L}_{tac}$ is way more powerful than tinycals, featuring constructs typical of high-level programming and defining their reduction semantics. However, since its aim was different, $\mathcal{L}_{tac}$ fails to address the interaction problem that tinycals do address.

Two alternative approaches for authoring structured HOL scripts have been proposed in [11] and [12]. The first approach, implemented in Syme's TkHOL, is similar to the one presented in this paper but lacks a formal description. Moreover, unlike HOL, we consider a logic with meta-variables which can be closed by side effects. Therefore the order in which branches are closed by tactics is relevant and must be

made explicit in the script. For this reason we support tinycals like "`focus`" and $i_1,\ldots,i_n$ "`:`" which were not needed in TkHOL. The second approach, by Takahashi et al., implements syntax directed editing by automatically claiming lemmata for each goal opened by the last executed tactic. This technique breaks down with meta-variables because they are not allowed in the statements of lemmata.

6 Conclusions

In this paper we presented the syntax and semantics of tinycals, a tactical language able to mimic some of the LCF tacticals so widespread in state-of-the-art proof assistants. Tinycals advantages over LCF tacticals is that their syntax is un-structured and their evaluation proceeds step by step, enabling the user to start execution of a structured script before its completion. Intermediate proof status can be inspected and tactics with side effects are supported as well. The neat result is better integration with user interfaces based on the CtCoq/Proof General paradigm. Some implementative issues have also been discussed, and the extension of the approach to other tacticals has been considered with negative results.

Tinycals have been implemented and are used in the Matita proof assistant for the ongoing development of its standard library. Users experienced with other proof assistants, in particular Coq, consider them a serious improvement in the proof authoring interface. This is not a big figure (our users are just the member of our research team at the time of writing), but is enough to motivate our work on them, hoping to see them adopted soon in other systems.

References

[1] Aspinall, D., *Proof General: A generic tool for proof development*, in: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, Lecture Notes in Computer Science **1785** (2000).

[2] Bertot, Y., *The CtCoq system: Design and architecture*, Formal Aspects of Computing **11** (1999), pp. 225–243.

[3] Delahaye, D., "Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du système Coq," Ph.D. thesis, Université Pierre et Marie Curie (Paris 6) (2001).
URL http://cedric.cnam.fr/~delahaye/publications/these-delahaye.ps.gz

[4] Geuvers, H. and G. I. Jojgov, *Open proofs and open terms: A basis for interactive logic*, in: J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CLS 2002*, Lecture Notes in Computer Science **2471** (2002), pp. 537–552.

[5] Gordon, M. J. C., R. Milner and C. P. Wadsworth, *Edinburgh LCF: a mechanised logic of computation*, Lecture Notes in Computer Science **78** (1979).

[6] *The Isabelle proof-assistant*,
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[7] Kirchner, F., *Coq Tacticals and PVS Strategies: A Small-Step Semantics*, in: *Design and Application of Strategies/Tactics in Higher Order Logics*, 2003.

[8] *The Mizar proof-assistant*,
http://mizar.uwb.edu.pl/.

[9] Muñoz, C., "A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory," Ph.D. thesis, INRIA (1997).

[10] *The NuPRL proof-assistant*,
http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html.

[11] Syme, D., *A new interface for hol - ideas, issues and implementation*, in: *Proceedings of Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, TPHOLs 1995*, Lecture Notes in Computer Science **971** (1995), pp. 324–339.

[12] Takahashi, K. and M. Hagiya, *Proving as editing HOL tactics*, Formal Aspects of Computing **11** (1999), pp. 343–357.

[13] The Coq Development Team, *The Coq proof assistant reference manual*, http://coq.inria.fr/doc/main.html (2005).

[14] Wenzel, M., *Isar - a generic interpretative approach to readable formal proof documents*, in: *Theorem Proving in Higher Order Logics*, 1999, pp. 167–184.

A Utility functions

The goal automatically selected by "[" or "|" is called *unhandled* until a tactic is applied to it. Unhandled goals are just postponed (not moved into the todo list τ) by $i_1,\ldots,i_n$"**:**". Goals opened by a tactic are marked with *mark_as_handled* to distinguishing them from unhandled goals. The function *renumber_branches* is used by "[" to name branches.

$$unhandled(l) = \begin{cases} true & \text{if } l = \langle n, \texttt{Open } g\rangle \wedge n > 0 \\ false & \text{otherwise} \end{cases}$$

$$mark\_as\_handled([g_1;\cdots;g_n]) = [\langle 0, \texttt{Open } g_1\rangle;\cdots;\langle 0, \texttt{Open } g_n\rangle]$$

$$renumber\_branches([\langle i_1, s_1\rangle;\cdots;\langle i_n, s_n\rangle]) = [\langle 1, s_1\rangle;\cdots;\langle n, s_n\rangle]$$

The next three functions returns open goals or tasks in the status or parts of it. Open goals are those corresponding to conjectures still to be proved.

$$get\_open\_tasks(l) = \begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Open } g\rangle :: get\_open\_tasks(tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \\ get\_open\_tasks(tl) & \text{if } l = hd :: tl \end{cases}$$

$$get\_open\_goals\_in\_tasks\_list(l) = \begin{cases} [\,] & \text{if } l = [\,] \\ g :: get\_open\_goals\_in\_tasks\_list(tl) & \text{if } l = \langle \_, \texttt{Open } g\rangle :: tl \\ get\_open\_goals\_in\_tasks\_list(tl) & \text{if } l = \langle \_, \texttt{Closed } g\rangle :: tl \end{cases}$$

$$get\_open\_goals\_in\_status(S) = \begin{cases} [\,] & \text{if } S = [\,] \\ get\_open\_goals\_in\_tasks\_list(\Gamma @ \tau @ \kappa) & \\ \quad @ get\_open\_goals\_in\_status(tl) & \text{if } S = \langle \Gamma, \tau, \kappa, \_\rangle :: tl \end{cases}$$

To keep the correspondence between branches in the script and ramifications in the proof, goals closed by side-effects are marked as `Closed` if they are in Γ (that keeps track of open branches). Otherwise they are silently removed from postponed goals (in todo list τ or dot continuation κ). `Closed` branches have to be accepted by the user with "`accept`".

$$close\_tasks(G, S) = \begin{cases} [\,] & \text{if } S = [\,] \\ \langle close_{aux}(G, \Gamma), \tau', \kappa', t\rangle :: close\_tasks(G, tl) & \text{if } S = \langle \Gamma, \tau, \kappa, t\rangle :: tl \\ \quad \text{where } \tau' = remove\_tasks(G, \tau) & \\ \quad \text{and } \kappa' = remove\_tasks(G, \kappa) & \end{cases}$$

$$close_{aux}(G, l) = \begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Closed } g\rangle :: close_{aux}(G, tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \wedge g \in G \\ hd :: close_{aux}(G, tl) & \text{if } l = hd :: tl \end{cases}$$

$$remove\_tasks(G, l) = \begin{cases} [\,] & \text{if } l = [\,] \\ remove\_tasks(G, tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \wedge g \in G \\ hd :: remove\_tasks(G, tl) & \text{if } l = hd :: tl \end{cases}$$

Web Interfaces for Proof Assistants

Cezary Kaliszyk[1]

Radboud University Nijmegen, The Netherlands

Abstract

This article describes an architecture for creating responsive web interfaces for proof assistants. The architecture combines current web development technologies with the functionality of local prover interfaces, to create an interface that is available completely within a web browser, but resembles and behaves like a local one. Security, availability and efficiency issues of the proposed solution are described. A prototype implementation of a web interface for the Coq proof assistant [8] created according to our architecture is presented. Access to the prototype is available on http://hair-dryer.cs.ru.nl:1024/.

Keywords: Proof Assistant, Interface, Web, Coq, Asynchronous DOM modification

1 Introduction

1.1 Motivation

Nowadays people are more and more accustomed to having a connection to the Internet all the time. Thus the network becomes a part of the computer one uses. As a consequence a tendency has emerged to provide services available just by accessing certain web pages. In this way people do not themselves need to install software for such services on their computers any more. Examples include web interfaces to e-mail, calendars, chat clients, word processors and maps.

Commercial services are often available through web-interfaces. On the other hand, in the scientific domain, examples are not so abundant. In particular there are no real implementations of web interfaces for proof assistants.

To use a proof assistant, one needs to install some software. Often the installation process is complicated. For example to install Isabelle [17], which is one of the most popular proof assistants, on a Linux system, one needs a particular version of PolyML, a HOL heap and Isabelle itself. To use an interface to access the prover, one needs ProofGeneral [4] and one of the supported Emacs versions. With Debian we had to downgrade the Linux kernel to support PolyML. The process described above is already complicated, not to mention other operating systems

[1] Email: cek@cs.ru.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and architectures, additional desirable patches and libraries, or less commonly used provers.

This is a problem. It happens that computer scientists prefer to stick with installed old versions of provers, not to go through the same process to upgrade. Mathematicians may even stay away from computer assisted proving, just because of the complexity of installation.

We want a fast interface, that is available with just a web browser. We want to access various proof assistants and their versions, in a uniform manner, without installing anything, not even plugins. The interface should look and behave like local interfaces to proof assistants.

We want the possibility to create web pages, that show tutorials and proofs, but that are bound to the prover itself, where the user can interact with the real system. The provider of the server may install patched versions of provers, allowing an easy way for the users to try out their features. We want libraries for proof assistants to be available centrally, so that users who want to see them do not need to download or install anything. The interface should allow developing proofs and libraries centrally, in a *wiki*-like [11] way.

1.2 Our Approach

The solution is a client-server architecture with a minimal lightweight client interpreted by the browser, a specialized HTTP server and background HTTP based communication between them. The key element of our architecture is the asynchronous DOM modification technique (sometimes referred to as *AJAX - Asynchronous JavaScript and XML* or *Web application*). The client part is on the server, and when the user accesses the interface page, it is downloaded by the browser, which is able to interpret it without any installation.

The user of the interface, accessing it with the browser, does not need to do anything when a modification is done on the server. Every time the user accesses a prover, the version of the prover that is currently installed on the server is used. The user can access any of the provers installed on the server, even a prover which does not work on the platform from which the connection is made.

Saving the files on the central server allows accessing them from any location, by just accessing the interface's page with a web browser. A central repository simplifies cooperation in proof development, by replacing versioning systems like CVS, which keeps a remote and a local copy, by a *wiki-like* mechanism, where the only copy is the remote one.

Our approach is presented as an architecture to create web interfaces to proof assistants, but it is not limited to them. The problems solved are relevant to creating web interfaces programs that have a state, include an *undo* mechanism, and their interfaces can be buffer oriented. Our architecture may be applied for example to buffer oriented programming languages, like Epigram [15].

1.3 Related work

There have been some experiments with providing remote access to a prover. None of them allowed efficient access without installing additional software.

LogiCoq [18] is a web interface to Coq [8]. It offers a window where one can insert the contents of whole Coq buffer and submit them for verification. It sends the whole buffer with standard HTTP request and refreshes the whole page. Therefore one can work efficiently only with tiny proofs.

The web interface to the Omega system [7], requires the Mozart interpreter to be installed on the user's machine. The use of the web browser is minimal, the whole interface is written in Mozart. Installation of Mozart is possible only for certain platforms which also makes the solution limited.

There are Java applets having built-in proof assistant functionality. Examples may include G4IP [19] or Logic Gateway [12]. The installation of a browser plug-in to support Java is not simple in a Unix environment and limiting provers to Java applets is undesired.

Web interfaces related to proof assistants and displaying mathematics on the web are worth mentioning. In particular:

- Helm [3] - (Hypertextual Electronic Library of Mathematics) A web interface that allows visualisation of libraries available for proof assistants.
- Whelp [2] - A content based search engine for finding theorems in proof assistants libraries, that supports queries requiring matching and/or typing.
- ActiveMath [16] - A web-based framework for learning mathematics that uses Java applets to communicate with a central server using OMDoc [13].

There are some commercial web interfaces and frameworks that use asynchronous DOM modification in non scientific domains.

The novelty of our architecture in comparison with existing web interfaces for theorem provers is that it allows the creation of an interface to a prover, that can look and behave very much like the ones offered by state-of-the-art local interfaces, but is available just by accessing a page with a web browser without installing any additional software, not even plugins. Because of the architecture, the network used to transfer information does not slow down the interaction. The idea to use asynchronous DOM modification to create an interface to a proof assistants has never been applied before.

1.4 Contents

In the rest of the paper we present the techniques for creation of web interfaces, that we will use (Section 2) and the internals of a local prover interfaces which we try to imitate (Section 3), followed by the presentation of the new architecture (Section 4) and a description of its security and efficiency (Section 5). We present our implementation prototype (Section 6). Finally we conclude and present a vision of future work (Section 7).

2 The Concept of Asynchronous DOM Modification

As the web is becoming more commonly used, web page designers and browser implementers add new functionality to web pages. Text files have been replaced by hyper-linked files, later including images, language-specific and mathematical

characters, styles and dynamic elements. The W3C Consortium, which is the organization responsible for the standardization of the Web, defines these elements as standards, and in consequence they are implemented in a similar ways in all browsers.

Since the late nineties browsers have started supporting the following technologies relevant to our research: JavaScript, DOM [14] and XmlHttp [20]. Combined use of these three technologies has became popular in recent years, since they allow one to create responsive web interfaces. In this document we refer to the combined usage of these three technologies as "*Asynchronous DOM Modification.*" One can find other names describing this technique, like *AJAX* or *Web Application.*

JavaScript is a scripting programming language, created by Netscape in 1995, for adding certain dynamic functionality to pages written in HTML. It has been quickly adopted by most browsers and nowadays it is supported even by some text mode browsers like w3m and Links, and mobile phone browsers. It is very often used on Internet websites.

DOM (Document Object Model) [14] is an API (Application programming interface) for managing HTML and XML documents that allows modifications of their structure and content. Recent browsers support W3C DOM accessibility by JavaScript. It is often used on web pages to add dynamic elements, for example drop-down menus or images that change when the mouse moves over them.

XmlHttp [20] is an API accessible by web browser scripting languages to transfer data to and from a web server. It internally uses HTTP requests. XmlHttp requests are sent to the server without the knowledge of the user of the web browser. For every XmlHttp request a callback has to be provided, to be executed when the response from the server is received. The sending of the request can be optionally asynchronous. XmlHttp has been available in most browsers for some time, and has been recently described in a W3C specification draft.

Asynchronous DOM modification is a web development technique that uses the three technologies described above to create responsive web interfaces. Such interfaces are web pages, where particular events (key presses and mouse movement) are captured by JavaScript events. The minimal client part encoded in JavaScript processes the local events, like menu opening or typing in a buffer. Events that require additional information from the server are sent in asynchronous XmlHttp requests. Since the request is done in the background, it does not interrupt the user from working locally. When the response arrives, it is used to modify the DOM of the page.

In comparison with classical web pages, the usage of asynchronous DOM modification makes it possible to send minimal information to the server, to receive only the information required, and to refresh small parts of the web page. Network overhead and page refreshing are minimized, thus creating interfaces which work many times faster then classical web-based ones. This way, the interface can closely resemble local interfaces, if network latency is reasonable. In case of high network latency, asynchronous requests allow the user to work locally, while additional data is requested.

Examples of usage of the asynchronous DOM modification are: webmails and calendars which operate within a single page, maps which download required parts

as they are dragged, and web chat clients. Such web interfaces are supported by all standard web browsers, in particular all Gecko based browsers, Microsoft Internet Explorer versions from 5, Opera from version 8, Konqueror from version 3.2, Safari from version 1.2 and even Nokia S60 browser from version 3. It is not supported by text mode browsers and browsers for visually impaired people.

3 Generic Interface for Proof Assistants

In this section we describe the internals of local interfaces for proof assistants. We chose for this ProofGeneral [4] for two reasons. First, it is a prover-independent interface to proof assistants. Second, it is popular, since it is universal and since it is built on the highly configurable Emacs text editor.

ProofGeneral's interface provides the user with two buffers: an editable buffer containing the proof script and the prover state buffer. ProofGeneral relies on the proof assistant to process the commands incrementally. It does not distinguish tactic-mode proofs from declarative-mode proofs. State changing and non-state-changing Coq commands are distinguished to make only the relevant ones part of the proof script and to allow queries.

The interface colors keywords according to the above distinctions, and additionally marks parts of the buffer with a background color, to indicate the status of verification. Possible states include: Expression that has been accepted by the prover, expression that is now being verified, and editable non-verified expression.

ProofGeneral provides a proof replying mechanism. The prover itself has to provide an *undo* mechanism. Users may choose a point in the buffer to go to, and ProofGeneral issues a number of proof steps and *undo* steps to the prover in order to reach that point.

ProofGeneral is responsible for providing the proof script from files on the disc to the prover and saving the buffers state. Other disc operations that exist in some provers, like proof compilation, program extraction or automated creation of documentation are not handled by ProofGeneral.

The current version of ProofGeneral is implemented mostly in Emacs Lisp, and is strongly tied with the editor itself. It is easy to adapt ProofGeneral to new proof assistants, by setting a number of variables. If this is not sufficient ELisp code can be used.

Other interfaces to provers offer mostly similar functionality. In some interfaces, like PCoq [1] or IsaWin, additional visualisation mechanisms are available, for example *term annotations*. Some of these mechanisms are not available in ProofGeneral; this limitation comes from the Emacs editor.

4 General Architecture

The two core elements of our architecture are: a specialized Web server and a communication mechanism (Fig. 1).

The Web server serves normal files and it is able to respond to special HTTP requests (see 4.2). The main interface is available as a normal HTML file on the server. When a user accesses the page with a browser, the page requires certain

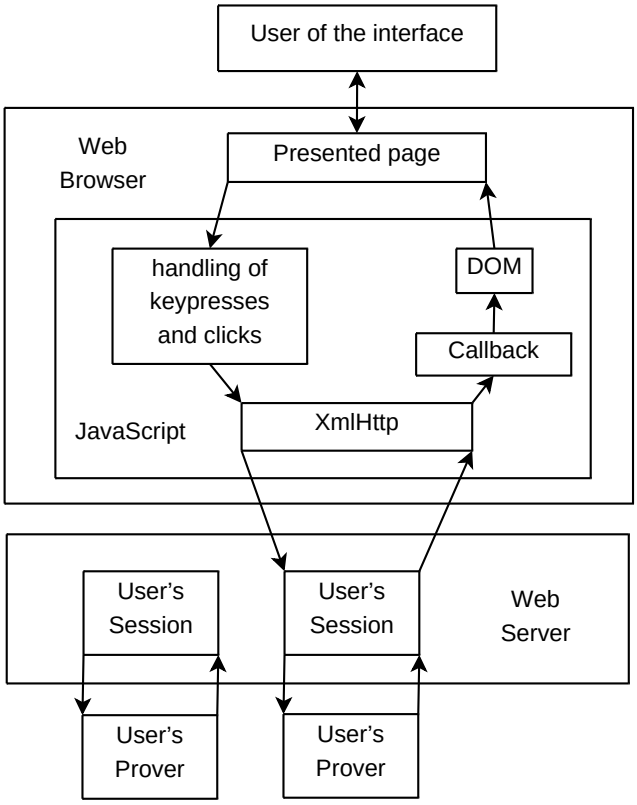


Fig. 1. General architecture.

JavaScript files, which are then downloaded and interpreted by the browser. This serves as the client part.

The communication between the client part and the server is done with the mechanism described in Section 2. HTTP requests are created in the background. The results are used to update the page in place. Only a small amount of information is transferred between the client and the server. The transfer is done asynchronously, making the interface responsive.

4.1 The Client Part

The client part offers a web page that initially presents the user with an editable buffer and an empty response buffer. (Also a menu or a toolbar is necessary for interaction, but they are normal elements of web pages). Buffers are implemented as HTML IFrames[2]. All keys that modify the IFrame are assigned to a special function. Locking of parts of the buffer is implemented by disallowing changes to locked parts of the buffer in this function.

When the user wants to verify a part of the buffer, this part is locked and sent to the server. Since the request is a background one, even if it takes a moment the user may continue working. When the response arrives, the contents of the two buffers are modified. The response may be a success, and then the part of the editable buffer is marked 'verified' and the response buffer shows the new prover state. If

[2] An *IFrame* is an HTML tag that includes a floating frame within a page, that can be optionally editable.

the command failed, the part of editable buffer is unlocked and the error shown. Parts of the editable buffer are marked, as their state changes, by using background colors, as it is done in ProofGeneral.

The interface includes a proof replaying mechanism, created in a similar way that it is done in local interfaces. When the user wants to go to a particular place in the buffer, this information is passed to the server. The server sends the commands to the client's prover session and informs the interface about the results. In a similar way the interface includes a *break* mechanism that allows stopping the prover's computation.

The interface includes functionality for file interaction. Files can be loaded and saved on the server. For interoperability downloading files and uploading files from the local computer may be provided. For proof development efficiency, insertion of templates and queries may be provided.

4.2 The Server Part

The server includes standard HTTP file serving functionality. With it the user's browser downloads the client part. The server can also handle special messages available for users, that have logged in. Session mechanism is used to support multiple clients. A session is created when a user logs in to the system and is sustained with a cookie mechanism. Every user's session is associated with a particular prover session. The server runs provers as subprocesses and communicates with them through standard input and output. Prover sessions are terminated after a long period of inactivity (if the user did not close the page, the client part can replay the proof script from the beginning).

The special messages, mentioned above, include: passing a given complete expression to verify to the prover, issuing an *undo* command in the prover, saving a file, loading a file, and *break* (stopping the prover computation). The commands from the client for the prover are passed first to the server, which transmits them to the prover. Prover replies are analysed by the server and only state changes are sent to the client. The state changes consist of two parts: changing of the markings of the edit buffer and the new contents of the prover state buffer.

Replies from the server are passed back to the client in an asynchronous way. This means, that the server does not answer HTTP requests from the client immediately, but when an answer from the prover is received or a timeout is reached. The server keeps a pool of provers that have been asked to process data, and waits for an answer from any of them. The waiting process does not block the server, that is, other clients' requests can be processed in the meantime.

5 Security and Efficiency

5.1 User side

All code that the user runs is interpreted within the web browser. Thus a malicious or virus infected prover can influence the client only by exploiting system or browser errors.

The efficiency of code execution on the user's side is dependent on the efficiency

of the browser's internal web page and scripts interpretation, and the speed of HTML rendering.

Our experiments show that client-side DOM changes with Internet Explorer are approximately twice as fast as with Mozilla Firefox (still usually invisible for the user). It is hard to say whether this is due to less security checks or the worse quality of the rendered page (no anti-aliasing) in Internet Explorer.

5.2 Server side

In any centralized environment security, availability and efficiency of the server are important. Standard security measures include a backup server prepared to take over network traffic in case of a primary server failure and regular backing up of user files. In this subsection we will describe only the issues and solutions particular to a server that runs a web interface to a prover.

Three kinds of issues arise: security, availability and equal sharing of resources. First, exploiting bugs in our architecture could lead crackers to take control of the server. Second, in a centralized environment the only copy of files is on the server. Unavailability of the server makes users not only unable to work, but also unable to access their files. Last, when users access the same server its resources are shared. If a particular prover uses all the memory or CPU, other users are unable to work.

To provide security, the server is run in a chrooted[3] environment, as a non-privileged user. The permissions include only reading server files and executing the provers. Every prover type is run as a different user (using the file `setuid` mechanism), that has read rights only to the prover's library, and write rights only to a directory where the prover's proof scripts are stored. To disallow storing overly large amounts of data, filesystem quota may be used.

For provers that allow system interaction, this functionality can be sometimes disabled. In particular, for ML based provers, dropping to the toplevel can be disabled. If the server administrator doesn't trust the prover's implementation, a secure version of the kernel can be used to disable irrelevant system calls. In this case even a language that is implemented inside ML can be available without changes to the prover itself.

To ensure equal sharing of resources, prover processes can be run with Cpu quota and memory quota mechanisms. The scheduling policy can be changed (for example with the `nice` system call) to provide the server process with priority over prover processes. Different provers have different CPU and memory requirements, which should be taken into account while setting the limits.

When many users want to access the interface, the resources of a single server may be insufficient. It is simple to run the server on a set of machines, by calling provers as subprocesses through `ssh` on separate computers. A load balancing mechanism can be implemented.

The communication between the server part and the client part can be secured by providing the interface through HTTPS.

[3] `chroot` is a system call preventing a process to access any files outside of a special root directory.

6 Implementation of a Prototype

We have implemented a minimal prototype of a web interface that follows the proposed architecture. The interface allows using the Coq proof assistant [8] with just a web browser, but it looks and behaves (Fig. 2) like the ones offered by CoqIde and ProofGeneral.

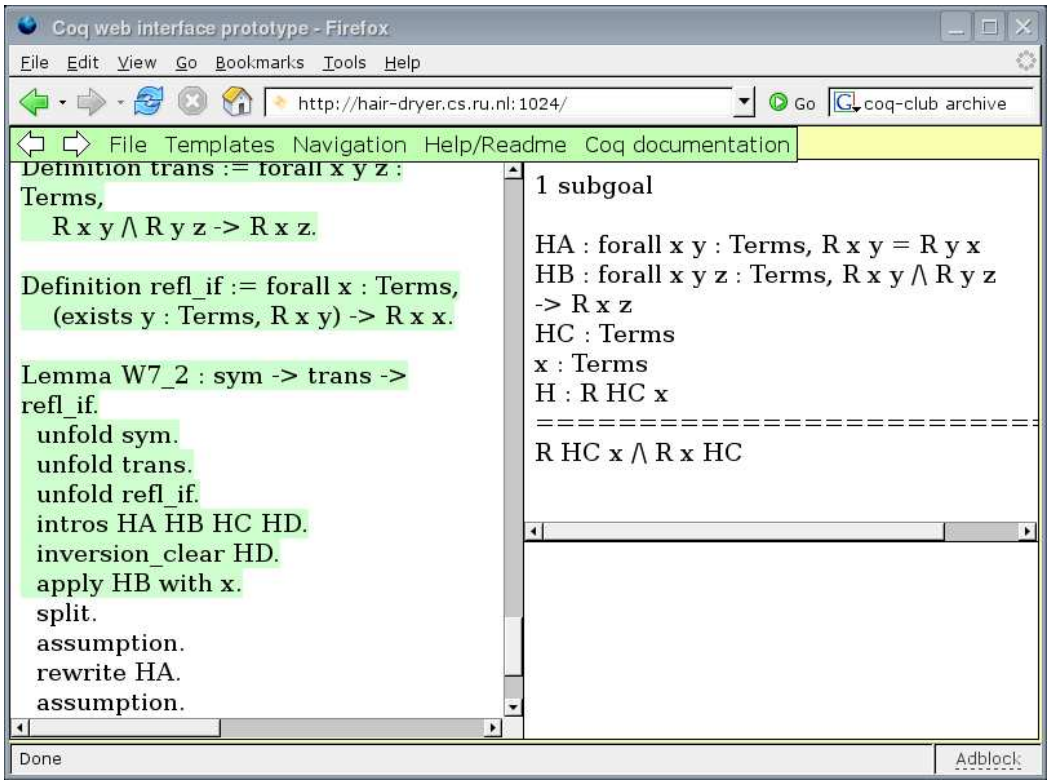


Fig. 2. Screenshot of the prototype, that shows working with a Coq proof. The verified part of the edit buffer is colored and locked. The state buffer shows the state of the proof, there are no Coq warnings.

Our server is a 400 line OCaml program, that serves two HTML files and a number of JavaScript files. It additionally supports special POST requests for verifying and for undoing commands as well as for loading and saving of files. It uses the OCamlHttpd library, for web-server functionality.

Our client consists of 10kB of JavaScript and 2kB of HTML. Most of the client-side code is responsible for the locking of the buffer and recognition of Coq expressions.

To secure our prototype the server is run as `nobody` in a minimal `chroot`ed environment. The prover sub-processes are `reniced` not to interfere with the main server process. Dropping from Coq to OCaml toplevel is disabled. The access to the interface is password protected, to avoid creating prover sessions for web-spiders. Web spiders are able only to see the saved proof scripts.

Our prototype includes a 1kB file, that is supposed to create a uniform layer that works with different browsers. We have not yet made it as general, as the asynchronous DOM modification is. In particular our prototype works well with Gecko-based browsers (Mozilla, Firefox, Galeon, ...). It works with Internet Ex-

plorer 6 and Opera 9, with some key-bindings missing (these browsers have them assigned to internal functions). It does not yet work with KHTML based browsers (Konqueror, Safari) and older versions of the above. We have tested our implementation's efficiency, by trying to use the server from other locations. Although it is hard to measure responsiveness to user's actions objectively, our experiments show, that with reasonable network latency, its responsiveness is very good.

The prototype is a Coq web interface, but there is not much code specific to Coq. The client part includes recognition of Coq comments and whole expressions to send. The server part includes recognition of successes and failures as well as the *undo* mechanism. For all ELisp code from ProofGeneral equivalent JavaScript regular expression handling can be provided. Thus adapting these three things to other provers should be simple, which is why we believe that implementing an interface according to our architecture that would support different provers can be easily done.

The client part has to overcome the minor differences between browsers. In particular it includes functions that create a uniform layer for XmlHttpRequest creation, event binding, and DOM that work the same way on all currently supported browsers.

6.1 Possible Uses

Our interface can be used to create interactive tutorials presenting proof assistants. We have created a special proof script, that includes a slightly modified version of the official Coq tutorial. The descriptive parts have been put inside comments (including the HTML formatting), and commands to the proof assistant have been left outside comments. A user that enters such a page may just read the tutorial and execute the commands in Coq environment, but may also do own experiments with it.

Non-trivial proof scripts that use tactics are unreadable without intermediate proof states. Thus proofs presented on the web are usually accompanied with some of the proof states usually automatically generated by Coqdoc or TeXmacs [6]. A web interface can be used (even in a read-only mode) to present such proofs interactively. In this way, the user reading the proof chooses which proof states to see.

External proof assistant libraries can be included on the server. With our server we included C-CoRN (Constructive Coq Repository at Nijmegen) [10]. Such libraries can be developed on the server. In such an approach visitors can always see and test the current version, without downloading and compiling the library.

Modified and experimental versions of provers usually require patching a particular version of the source of the proof assistant. Presenting such a modified version to others is easily possible with the given infrastructure. The server offered includes the Declarative Proof Language extension for Coq [9].

7 Conclusion

We presented an architecture to create simple, lightweight and fast web interfaces to proof assistants. Such interfaces are a novelty in the domain. Our solution works with modern web browsers without installing any additional software. The installation and updating process is done only on the server, the users do not need to do anything. It is therefore completely platform independent.

The communication mechanism makes the usage of the network minimal, therefore making the interface comparably responsive to local ones. In comparison with other client-server solutions, the only limitation is the dependency on the web browser. Fortunately web browsers include full scripting languages, allowing implementation of nearly all possible functionality of the interface on the client side. In particular the browser's internal editors are weak in comparison with local editors. One can implement in JavaScript the handling of more key bindings to make the editor similar to a local one. Most features of state-of-the-art local interfaces for proof assistants can be imitated this way. The efficiency of an editor implemented in JavaScript would depend on the browser interpreting it. We have not been able to find any such editor.

We believe that a centralized environment, with provers accessible through a web interface, is not limited in comparison with local interfaces, and that the architecture we have proposed is in the spirit of the current trends of development in computer science.

7.1 Future Work

Our primary focus is to extend the proposed architecture to a complete wiki-like architecture. This requires a versioning mechanism and merging of users' changes on the server. Additionally proof displaying and searching mechanisms are mandatory. Editing conflicts can be resolved in similar way as it is done in wiki software. For example if the file was changed and a user wants to save over it, differences are presented.

We would like to see how well our solution fits with the general prover interaction protocol PGIP [5]. The protocol is XML-based, so parts of it may even be passed by the server directly to browsers, since they are already able to parse XML. On the other hand the protocol may include too much information, since it was designed as a local one.

Finally we would like to create an implementation that includes all the features of our proposed architecture. Ideas include: providing other provers, making it compatible with all browsers that support asynchronous DOM modifications, implementing the *break* mechanism, compiling Coq files automatically, adding syntax highlighting, and providing better security.

References

[1] Amerkad, A., Y. Bertot, L. Pottier and L. Rideau, *Mathematics and proof presentation in PCoq*, Rapport de Recherche 4313, Inria, Sophia Antipolis (2001).

[2] Asperti, A., F. Guidi, C. S. Coen, E. Tassi and S. Zacchiroli, *A content based mathematical search engine: Whelp.*, in: J.-C. Filliâtre, C. Paulin-Mohring and B. Werner, editors, *TYPES*, Lecture Notes in Computer Science **3839** (2004), pp. 17–32, URL: http://www.bononia.it/~zack/stuff/whelp.pdf.

[3] Asperti, A., L. Padovani, C. S. Coen and I. Schena, *Helm and the semantic math-web.*, in: R. J. Boulton and P. B. Jackson, editors, *TPHOLs*, Lecture Notes in Computer Science **2152** (2001), pp. 59–74, URL: http://helm.cs.unibo.it/smweb.ps.gz.

[4] Aspinall, D., *Proof General: A generic tool for proof development.*, in: S. Graf and M. I. Schwartzbach, editors, *TACAS*, Lecture Notes in Computer Science **1785** (2000), pp. 38–42.

[5] Aspinall, D., C. Lüth and D. Winterstein, *A framework for interactive proof*, in: D. Aspinall, editor, *Proceedings of the ETAPS-05 Workshop on User Interfaces for Theorem Provers (UITP-05), Edinburgh*, 2005, p. 15, URL: http://proofgeneral.inf.ed.ac.uk/Kit/docs/pgipimp.pdf.

[6] Audebaud, P. and L. Rideau, *Texmacs as authoring tool for formal developments.*, Electr. Notes Theor. Comput. Sci. **103** (2004), pp. 27–48.

[7] Benzmüller, C., L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. H. Siekmann and V. Sorge, *Omega: Towards a mathematical assistant.*, in: W. McCune, editor, *CADE*, Lecture Notes in Computer Science **1249** (1997), pp. 252–255.

[8] Coq Development Team, "The Coq Proof Assistant Reference Manual Version 8.0," INRIA-Rocquencourt (2005), URL: http://coq.inria.fr/doc-eng.html.

[9] Corbineau, P., *Declarative proof language for Coq* (2006), URL: http://www.cs.ru.nl/~corbineau/mmode.en.html.

[10] Cruz-Filipe, L., H. Geuvers and F. Wiedijk, *C-CoRN, the constructive Coq repository at Nijmegen.*, in: A. Asperti, G. Bancerek and A. Trybulec, editors, *MKM*, Lecture Notes in Computer Science **3119** (2004), pp. 88–103.

[11] Davies, J., "Wiki Brainstorming and Problems with Wiki Based Collaboration," Master's thesis, University of York (2004).

[12] Gottschall, C., *Logic gateway* (2005), URL: http://logik.phl.univie.ac.at/~chris/gateway/.

[13] Kohlhase, M., *Omdoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge.*, in: J. A. Campbell and E. Roanes-Lozano, editors, *AISC*, Lecture Notes in Computer Science **1930** (2000), pp. 32–52.

[14] Le Hors, A., P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion and S. Byrne, *Document Object Model (DOM) Level 3 Core Specification, Version 1*, W3C Recommendation (2004).

[15] McBride, C., *Epigram: Practical programming with dependent types.*, in: V. Vene and T. Uustalu, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science **3622** (2004), pp. 130–170.

[16] Melis, E., J. Büdenbender, G. Goguadze, P. Libbrecht and C. Ullrich, *Knowledge representation and management in activemath.*, Ann. Math. Artif. Intell. **38** (2003), pp. 47–64.

[17] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL - A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science **2283**, Springer, 2002.

[18] Pottier, L., *LogiCoq* (1999), URL: http://wims.unice.fr/wims/wims.cgi?module=U3/logic/logicoq.

[19] Urban, C., *Implementation of proof search in the imperative programming language pizza.*, in: H. C. M. de Swart, editor, *TABLEAUX*, Lecture Notes in Computer Science **1397** (1998), pp. 313–319.

[20] Web APIs Working Group, *The XMLHttpRequest Object*, Technical report, W3C (2006), URL: http://www.w3.org/TR/XMLHttpRequest/.

PLATΩ: A Mediator between Text-Editors and Proof Assistance Systems

Marc Wagner

Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.ags.uni-sb.de/~mwagner)

Serge Autexier

DFKI GmbH & Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.dfki.de/~serge)

Christoph Benzmüller

Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.ags.uni-sb.de/~chris)

Abstract

We present a generic mediator, called PLATΩ, between text-editors and proof assistants. PLATΩ aims at integrated support for the development, publication, formalization, and verification of mathematical documents in a natural way as possible: The user authors his mathematical documents with a scientific WYSIWYG text-editor in the informal language he is used to, that is a mixture of natural language and formulas. These documents are then semantically annotated preserving the textual structure by using the flexible, parameterized proof language which we present. From this informal semantic representation PLATΩ automatically generates the corresponding formal representation for a proof assistant, in our case ΩMEGA. The primary task of PLATΩ is the maintenance of consistent formal and informal representations during the interactive development of the document.

1 Introduction

Unlike computer algebra systems, mathematical proof assistance systems have not yet achieved considerable recognition and relevance in mathematical practice. Clearly, the functionalities and strengths of these systems are generally not sufficiently developed to attract mathematicians on the edge of research. For applications in e-learning and engineering contexts their capabilities are often sufficient, though. However, even for these applications significant progress is still required, in particular with respect to the usability of these systems. One significant shortcoming of the current systems is that they are not fully integrated into or accessible from within standard mathematical text-editors.

For purposes such as tutoring mathematics, communicating or publishing mathematical documents, the content is in practice usually encoded using common mathematical representation languages by employing standard mathematical text editors. Proof assistance

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

systems, in contrast, require fully formal representations and are not yet sufficiently linked with these standard mathematical text-editors. Therefore, rather than developing a new user interface for the mathematical assistance system ΩMEGA [24], we propose a generic way of extending ΩMEGA to serve as a mathematical service provider for scientific text-editors.

'If the mountain won't come to Mohammed, Mohammed must go to the mountain.'

Our approach allows the user to write his mathematical documents in the language he is used to, that is a mixture of natural language and formulas. These documents can then be semantically annotated preserving the textual structure by using the flexible parameterized proof language we present. From this semantic representation PLATΩ automatically builds up the corresponding formal representation in ΩMEGA and takes further care of the maintenance of consistent versions.

The formal representation allows the underlying proof assistance system to support the user in various ways, including the management of mathematical definitions, theorems and proofs, as well as the access to automatic theorem provers, computer algebra systems, and other mathematical tools in order to automatically verify conclusions and computations made by the user and to suggest possible corrections. These functionalities can be provided through PLATΩ by context-sensitive service menus in order to support the interactive development of mathematical documents at a high level of abstraction.

On the one hand, these services could include the possibility to automatically generate parts of the proof as well as computations in order to disburden the user of taking care about cumbersome details and to let him concentrate on the substantial parts of the proof. Thus, menu interaction may lead to changes of the formal representation which are reflected by PLATΩ in changes of the semantic representation in the document. On the other hand, further proof development in the text-editor leads to changes in the document which are propagated by PLATΩ to changes in the formal representation in ΩMEGA.

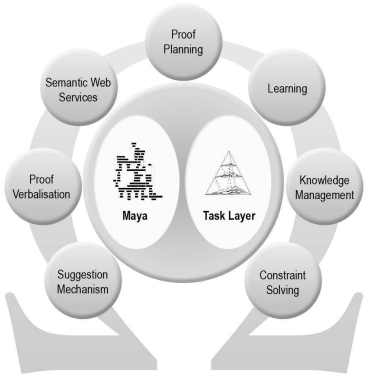
Altogether, this approach allows for the incremental, interactive development of mathematical documents which in addition can be formally validated by ΩMEGA, hence obtaining *verified mathematical documents*. This approach is generally independent of the proof assistance system as well as the text-editor. Nevertheless the scientific WYSIWYG text-editor TEXMACS [27] provides professional type-setting and supports authoring with powerful macro definition facilities like in LATEX. It moreover allows for the definition of plug-ins that automatically process the document and is thus especially well-suited for an integration of PLATΩ.

This paper is organized as follows: Section 2 presents an overview on the ΩMEGA system in order to more concretely motivate our setting. Section 3 introduces the mediator PLATΩ with a focus on the interfaces to the text-editor and the proof assistance system. A working example is presented in Section 4 that illustrates the integration of PLATΩ into a scientific text editor like for example TEXMACS. The paper concludes with an overview on related work (Section 5) and a summary of the major results in Section 6.

2 Preliminaries: ΩMEGA, MAYA, and the TASK LAYER

The development of the proof assistance system ΩMEGA is one of the major attempts to build an all encompassing assistance tool for the working mathematician or for the formal work of a software engineer. It is a representative of systems in the paradigm of *proof*

planning and combines interactive and automated proof construction for domains with rich and well-structured mathematical knowledge (see Figure on the right). The ΩMEGA-system is currently under re-development where, among others, it is augmented by the development graph manager MAYA and the underlying natural deduction calculus is replaced with the CORE-calculus [4].



The MAYA system [8] supports an evolutionary formal development by allowing users to specify and verify developments in a structured manner, it incorporates a uniform mechanism for verification in-the-large to exploit the structure of the specification, and it maintains the verification work already done when changing the specification. Proof assistance systems like ΩMEGA rely on mathematical knowledge formalized in structured theories of definitions, axioms and theorems. The MAYA system is the central component in the new ΩMEGA system that takes care about the management of change of these theories via its OMDOC-interface [19].

The CORE-calculus supports proof development directly at the *assertion level* [17], where proof steps are justified in terms of applications of definitions, axioms, theorems or hypotheses (collectively called *assertions*). It provides the logical basis for the so-called TASK LAYER [14], that is an instance of the new proof datastructure (PDS) [5]. The TASK LAYER is the central component for computer-based proof construction in ΩMEGA. It offers a uniform proof construction interface to both the human user and the automated proof search procedures MULTI [21] and ΩANTS [9,26]. The nodes of the PDS are annotated with *tasks*, which are Gentzen-style multi-conclusion sequents augmented by means to define multiple foci of attention on subformulas that are maintained during the proof. Each task is reduced to a possibly empty set of subtasks by one of the following proof construction steps: (1) the introduction of a proof sketch [30] [1] , (2) deep structural rules for weakening and decomposition of subformulas, (3) the application of a lemma that can be postulated on the fly, (4) the substitution of meta-variables, and (5) the application of an inference. Inferences are the basic reasoning steps of the TASK LAYER, and comprise assertion applications, proof planning methods or calls to external theorem provers or computer algebra systems (see [14,6] for more details about the TASK LAYER).

A formal proof requires to break down abstract proof steps to the CORE calculus level by replacing each abstract step by a sequence of calculus steps. This has usually the effect that a formal proof consists of many more steps than a corresponding informal proof of the same conjecture. Consequently, if we manually construct a formal proof many interaction steps are typically necessary. Formal proof sketches [30] in contrast allow the user to perform high level reasoning steps without having to justify them immediately. The underlying idea is that the user writes down only the interesting parts of the proof and that the gaps between these steps are filled in later, ideally fully automatically (see also [24]). Proof sketches are thus a highly relevant for a mediator like PLATΩ whose task it is to support the transition to fully formal representations from an informal proof in a mathematical document via intermediate representations of underspecified proof sketches.

[1] In the old ΩMEGA system this was realized by using so-called *Island*-methods.

Figure 1. PLATΩ mediates between natural mathematical texts and the proof assistant ΩMEGA

3 The PLATΩ System

The mediator PLATΩ is designed to run either locally as a plugin for a particular text-editor or as a mathematical service provider which text editors could access through the web. In order to manage different text-editor clients as well as different documents in the same client, we integrated session management into PLATΩ. The text-editor may request a unique session key which it has to provide as an argument for any further interaction in this particular session.

PLATΩ is connected with the text-editor by an informal representation language which flexibly supports the usual textual structure of mathematical documents. Furthermore, this semantic annotation language, called *proof language* (PL), allows for underspecification as well as alternative (sub)proof attempts. In order to generate the formal counterpart of a PL representation, PLATΩ separates theory knowledge like definitions, axioms and theorems from proofs. The theories are formalized in the *development graph language* (DL), which is close to the OMDOC theory language supported by the MAYA system, whereas the proofs are transformed into the *tasklayer language* (TL) which describes the PDS instance of the TASK LAYER. Hence, PLATΩ is connected with the proof assistance system ΩMEGA by a formal representation close to its internal datastructure.

Besides the transformation of complete documents, it is essential to be able to propagate small changes from an informal PL representation to the formal DL/TL one and the way back. If we always perform a global transformation, we would on the one hand rewrite the whole document in the text-editor which means to lose large parts of the natural language text written by the user. On the other hand we would reset the datastructure of the proof assistance system to the abstract level of proof sketches. For example, any already developed expansion towards calculus level or any computation result from external systems would be lost. Therefore, one of the most important aspects of PLATΩ's architecture is the propagation of changes.

The formal representation finally allows the underlying proof assistance system to support the user in various ways. PLATΩ provides the possibility to interact through context-sensitive service menus. If the user selects an object in the document, PLATΩ requests service actions from the proof assistance system regarding the formal counterparts of the selected object. Hence, the mediator needs to maintain the mapping between objects in the informal language PL and the formal languages DL and TL.

In particular, the proof assistance system could support the user by suggesting possible inference applications for a particular proof situation. Since the computation of all inference argument instantiations may take a long time, a multi-level menu with the possibility of lazy evaluation is required. PLATΩ supports the execution of nested actions inside a service menu which may result in a change description for this menu.

Through service menus the user may get access to automatic theorem provers and computer algebra systems which could automatically verify conclusions and computations and suggest possible corrections. These and many more functionalities are supported by PLATΩ through its mechanism to propagate changes as well as the possibility of custom answers to the user of the text-editor. Altogether, the mediator PLATΩ is designed to support the interactive development of mathematical documents at a high level of abstraction.

3.1 PLATΩ*'s Interfaces*

PLATΩ provides abstract interfaces to the text-editor and the proof assistance system (see also Fig. 1). Before we discuss their design and realization, we first present the functionalities of PLATΩ from the perspective of the text-editor. PLATΩ's methods are:

- **Initialize a session**: `plato:init` starts a new session in PLATΩ
- **Upload a document**: `plato:upload` uploads a whole document in the informal language PL, from which PLATΩ builds up the formal representations DL and TL. If a document has already been uploaded, PLATΩ performs an internal difference analysis using a semantic based differencing mechanism [22] and then proceeds as with patching the document.
- **Patch a document**: `plato:patch` patches an already uploaded document in the informal language PL with patch information given in the XUPDATE standard (see Section 3.2). PLATΩ transforms this patch information into patches for the formal representations DL and TL, which are used to patch the datastructure of the proof assistance system.
- **Request a menu**: `plato:service` requests a menu for an object in the informal language PL inside the document. The response is either a menu in the service language SL (or an error message). PLATΩ uses its maptable relating objects in PL with objects in DL and TL to requests service support from the proof assistance system for the latter.
- **Execute a menu action**: `plato:execute` triggers the execution of an action with its actual arguments. The result can be a patch for the current menu, a patch for the document or a custom answer (or an error message). The purpose is to evaluate an action inside a menu. This style of responses offers quite many interaction possibilities: If the selected action was nested somewhere in the menu, the proof assistance system will usually modify the menu. This will be propagated by PLATΩ to a corresponding response which only modifies the menu and leaves the patch for the document and the custom answer empty. If the selected action was situated on top level of the menu, the execution in the proof assistance system will more likely change the formal representation. Anyhow, PLATΩ propagates these changes to changes in the informal presentation of the text-editor, such that the response will usually remove the menu and patch the document appropriately. The custom answer leaves room for arbitrary interaction possibilities like knowledge retrieval or natural language feedback.

- **Close a session**: `plato:close` terminates a session.

A detailed descriptions of PLATΩ's interface functions is given in Appendix A.

3.2 Interface to the Text-Editor and Proof Assistance System

The goal of PLATΩ is to lay a compatible foundation for a text-editor interface across different environments. It should be a clean, extensible interface that is very simple and easy to implement such that it could quickly be adapted to run with any scientific text editor on any operating system. Therefore we decided to represent the mathematical document as well as the service menus in XML [11], the patches for documents and menus in the XUPDATE update language [20] and to use XML-RPC [31] as interface protocol. XUPDATE [20] is an XML update language which uses XML to encode its updates and the expression language XPATH [10] to select elements for processing. An update may contain the following types of elements relevant for PLATΩ: `insert-before`, `insert-after`, `append`, `update`, `remove`. All operations in an update have to be applied in parallel to the target document. XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism. It is a very simple protocol, defining only a handful of data types and commands, and its entire two page description can be found at [31].

The ΩMEGA system is implemented in LISP. Therefore, we decided to implement the interface to ΩMEGA, which provides LISP functions for each PLATΩ method, in LISP too. These functions operate only on the formal representation of the mathematical document and they will be illustrated in more detail in the next Section. PLATΩ allows to start, stop and manage multiple servers in parallel for the same proof assistance instance. Generally, we aim at an approach that is independent of the particular proof assistance system to be integrated. Therefore the proof language as well as the service menu language are parameterized over the sublanguages for definitions, formulas, references and menu argument content. Extending these sublanguages allows to scale up the power of the whole system regarding representation capabilities as well as service functionalities. As soon as there will be significant progress in the area of natural language analysis, one could even allow full natural language in these sublanguages. Thus PLATΩ is designed to support the evolution of the underlying proof assistance system towards an ideal mathematical assistance system. We will present some more aspects of this more general viewpoint in the next Section. The focus, however, is on the integration of the ΩMEGA system into the scientific WYSIWYG text-editor TEXMACS.

4 A Working Example

In this section we will evaluate the mediator PLATΩ in combination with ΩMEGA and TEXMACS. We will illustrate all available methods of PLATΩ by discussing a working example in the theory of Simple Sets.

In this paper, we describe the mediation between the informal representation in the text-editor and the formal representation in the proof assistance system on an abstract level. All details on the communicated documents, patch descriptions and menus for this example can be found in [28].

Since the TEXMACS interface for proof assistance systems is under continuous develop-

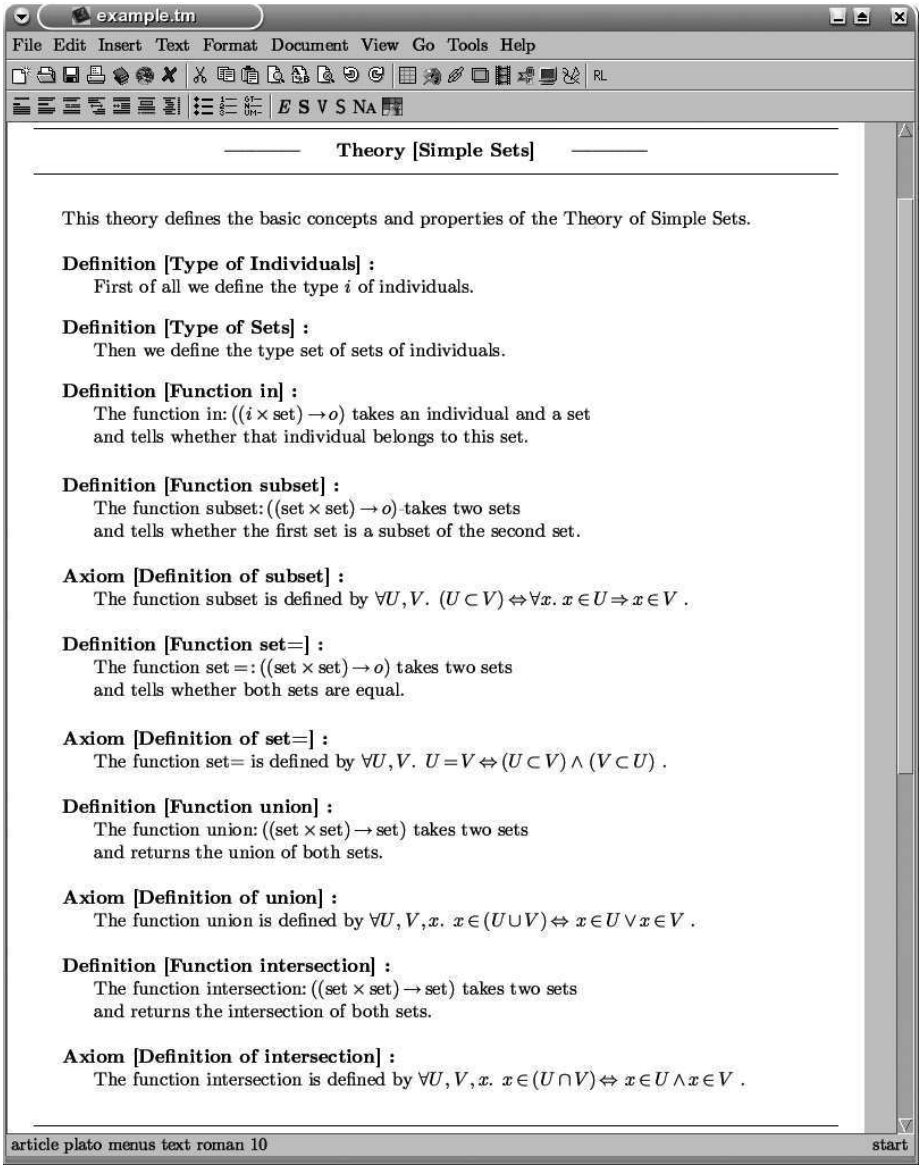


Figure 2. Theory *Simple Sets* in TEXMACS

ment, a PLATΩ plugin for TEXMACS has been developed by the ΩMEGA group that maps the interface functions of PLATΩ to the current ones of TEXMACS and which defines a style file for PL macros in TEXMACS. In the following example, we use this plugin to establish a connection between TEXMACS and PLATΩ's XML-RPC server.

First of all, the text-editor TEXMACS initializes a new session by calling the method `plato:init` together with a client name, for example "`texmacs#1`". The resulting session name has to be saved by the text-editor in order to use it for the following communication with PLATΩ.

In the text-editor, we have written an example document with the semantic annotation language PL (defined in [28]). The theory *Simple Sets* in this document contains for example definitions and axioms for *subset*, *set=*, *union* and *intersection*. Fig. 2 shows the theory as seen in TEXMACS and Fig. 3 shows the encoding of this theory in TEXMACS with PLATΩ

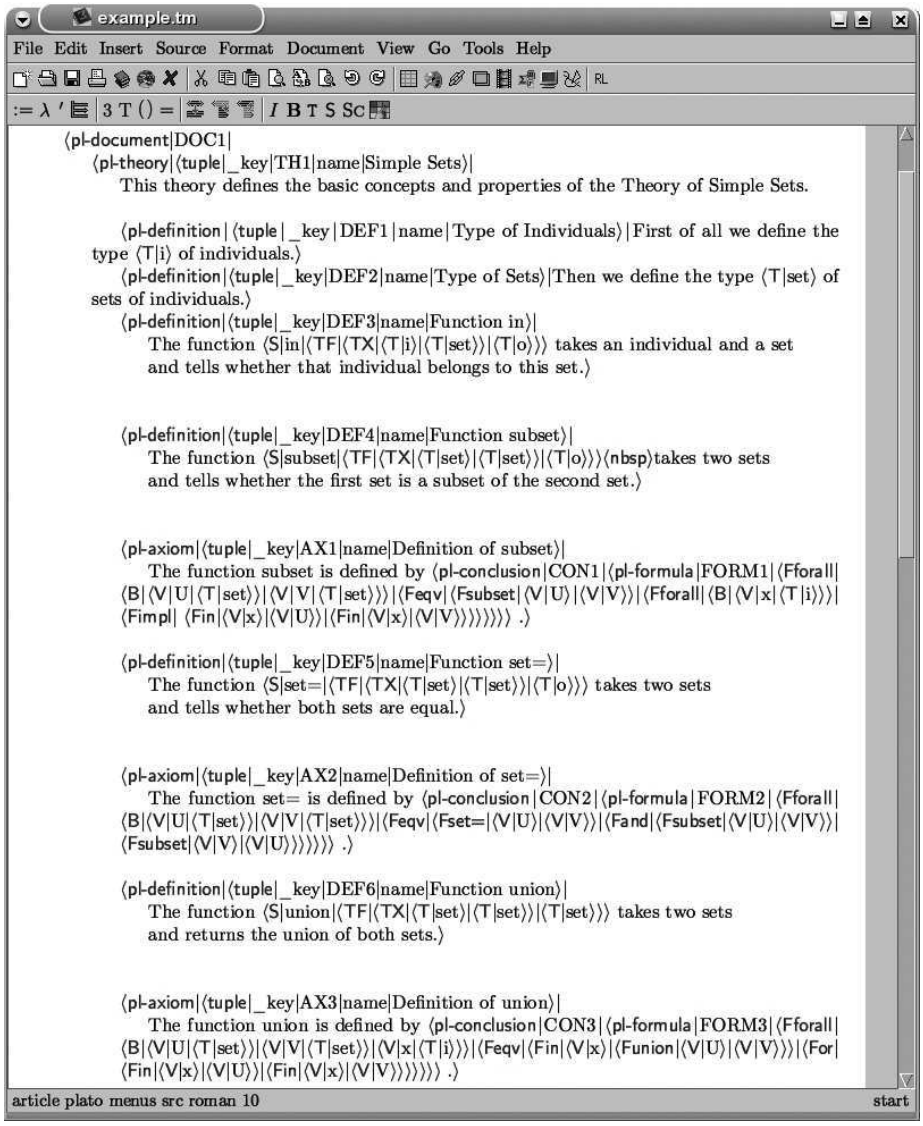


Figure 3. Encoding of Theory *Simple Sets* in TEXMACS

macros.

Furthermore, we have written a theory *Distributivity in Simple Sets* which imports all knowledge from the first theory *Simple Sets*. This second theory consists of a theorem about the *Distributivity of intersection*. The user has already started a proof for this theorem by introducing two subgoals. Fig. 4 shows the theory as seen in TEXMACS and Fig. 5 shows the encoding of this theory in TEXMACS.

By pressing a keyboard shortcut, the user can always easily switch between both views in the text-editor. The PL macros contained in the document must be provided by the user [2] and are used to automatically extract the corresponding PL document, the informal representation of the document for PLATΩ.

Uploading this PL document with `plato:upload`, PLATΩ separates theory knowl-

[2] Currently this still requires some expertise about PL and the TEXMACS macro language. Future work includes to provide better support for this task.

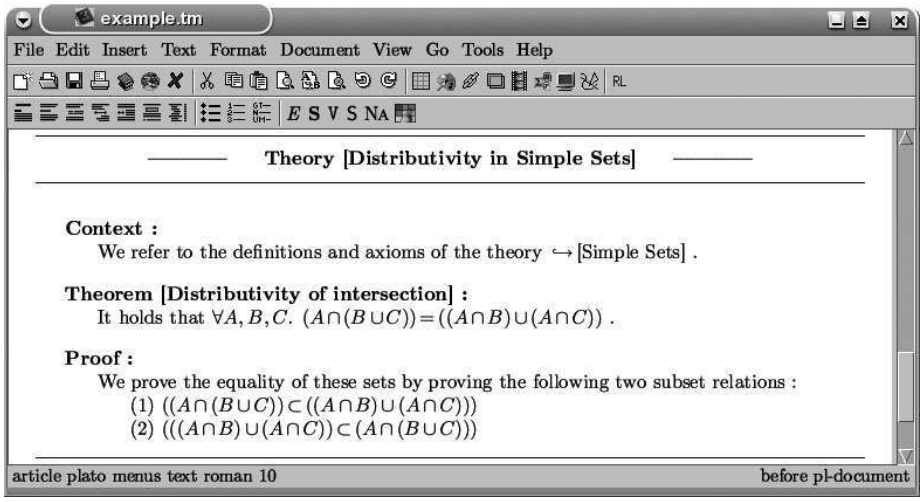


Figure 4. Theory *Distributivity in Simple Sets* in TEXMACS

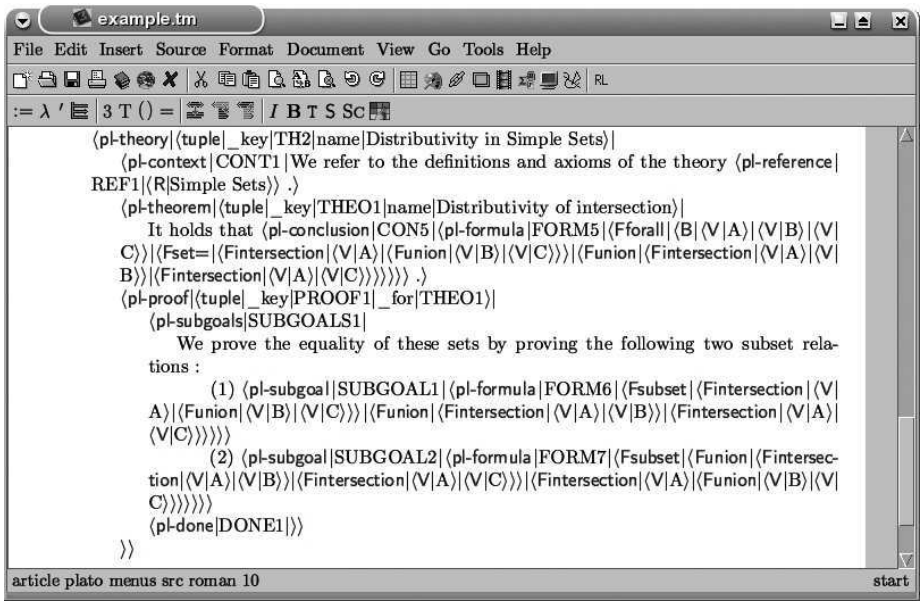


Figure 5. Encoding of Theory *Distributivity in Simple Sets* in TEXMACS

edge like definitions, axioms and theorems from proofs and starts generating the formal representation.

On the one hand, PLATΩ creates a DL document containing definitions, axioms and theorems in a representation close to OMDOC. On the other hand, the proof is transformed into a TL document, an abstract representation for the PDS instance of the TASK LAYER in the proof assistance system.

From the DL document, the PLATΩ instance for ΩMEGA generates a theory representation in OMDOC that MAYA takes as input for the creation of a development graph. Fig. 6 shows the theories uploaded in ΩMEGA. For this evaluation we use the old user interface LΩUI [25] to visualize the status of ΩMEGA. The user interacts of course only with the text-editor. The old LΩUI interface, including the display of MAYA's development graphs, shall be entirely replaced by TEXMACS and PLATΩ. They are only presented in this paper

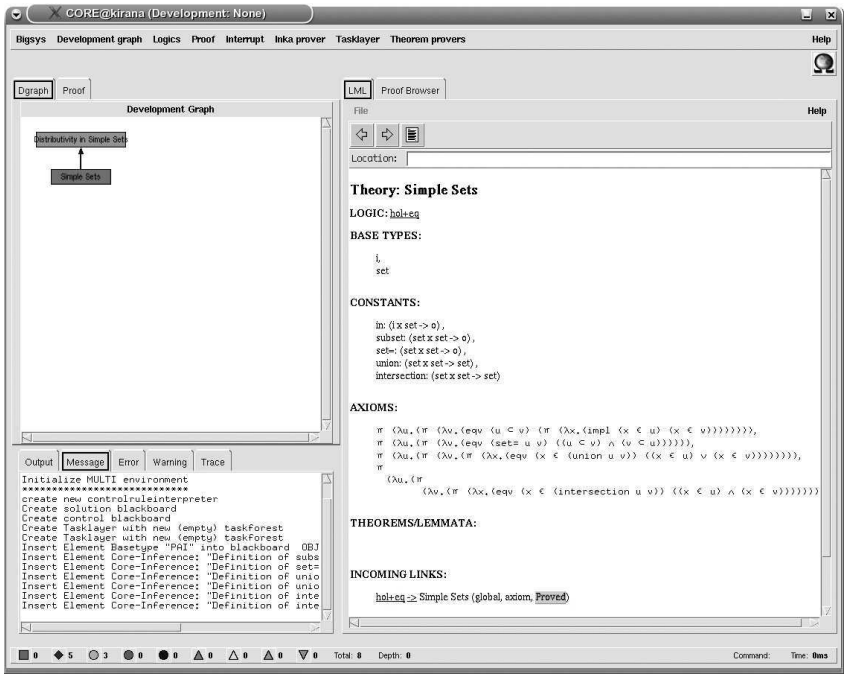


Figure 6. Theory *Simple Sets* in ΩMEGA

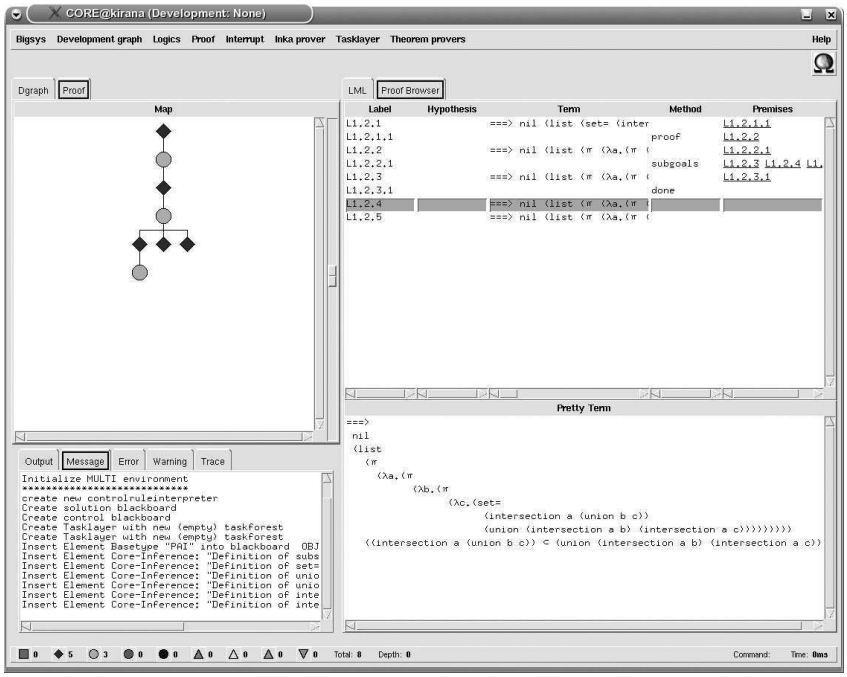


Figure 7. Partial Proof of Theorem *Distributivity of intersection* in ΩMEGA

to show the internal representation obtained from TEX_{MACS} via PLATΩ. From the TL document, the PLATΩ instance for ΩMEGA builds up the concrete datastructure of the TASK LAYER (see Fig. 7).

The upload procedure has terminated successfully with the complete generation of the formal representation in the proof assistance system, hence PLATΩ returns "OK".

Further developing the document, the user has started to prove the first subgoal by de-

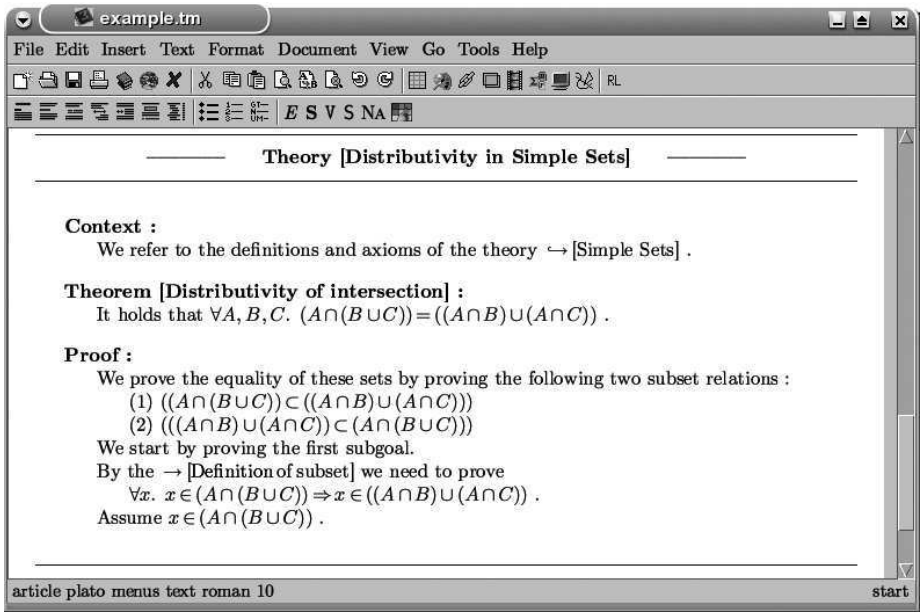


Figure 8. Modification of the Proof in TEX$_{MACS}$ by the User

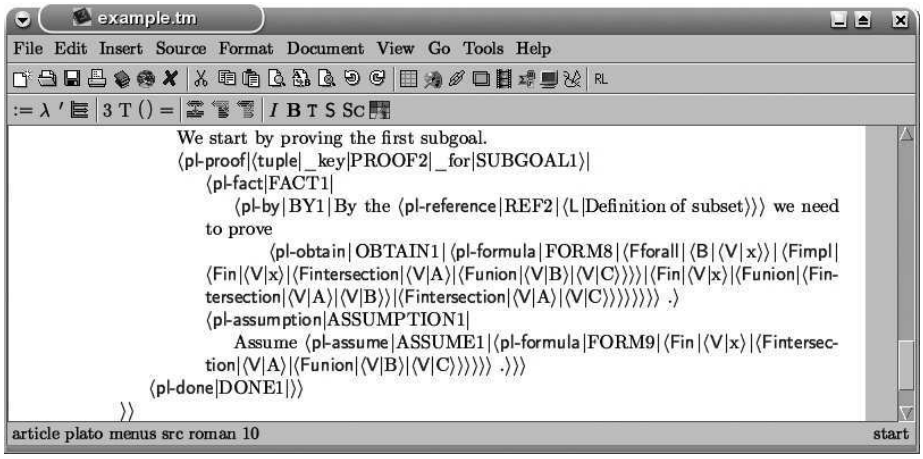


Figure 9. Modification of the Encoding of the Proof in TEX$_{MACS}$ by the User

riving a new subgoal and introducing an assumption (see Fig. 8). This modification of the encoding of the document (see Fig. 9) has to be propagated by PLATΩ to the formal representation in ΩMEGA. In general, the difference with respect to the last synchronized version of the document should be computed and send to PLATΩ by using `plato:patch`. At the moment, TEX$_{MACS}$ is not able to compute this difference, therefore the whole document is send again by `plato:upload` and PLATΩ computes the difference.

The difference of the informal PL document is then transformed by PLATΩ to a difference of the formal representations in DL and TL. Since the modifications do not affect theory knowledge, this transformation only results in modifications for the intermediate representation and finally the representation of the TASK LAYER proof data structure.

The PLATΩ instance for ΩMEGA uses this patch information to modify the TASK LAYER rather than to completely rebuild it from scratch (see Fig. 10). The patch procedure has terminated successfully, hence PLATΩ returns "`OK`". Altogether, the user is

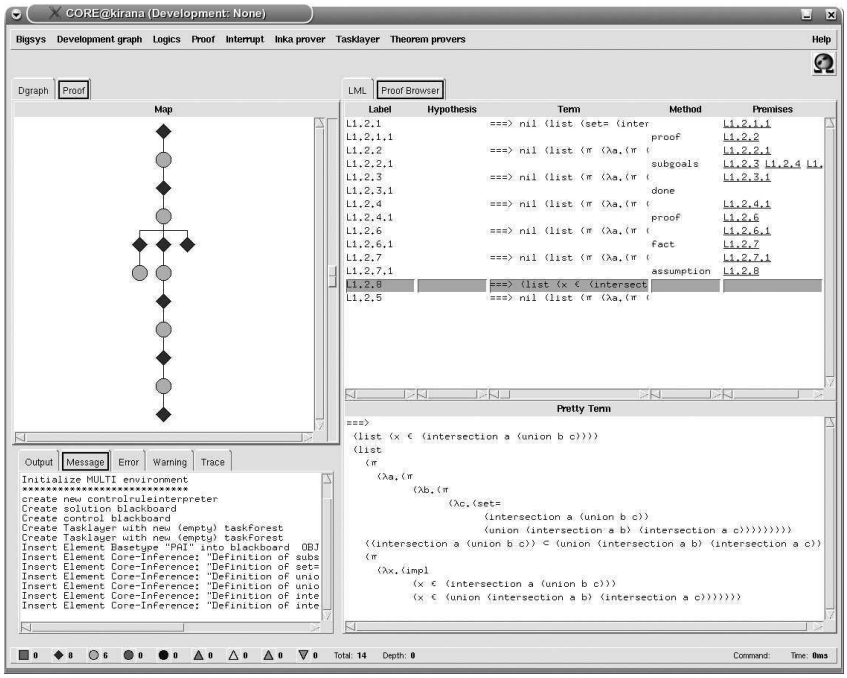


Figure 10. Modification of the Proof in ΩMEGA by PLATΩ

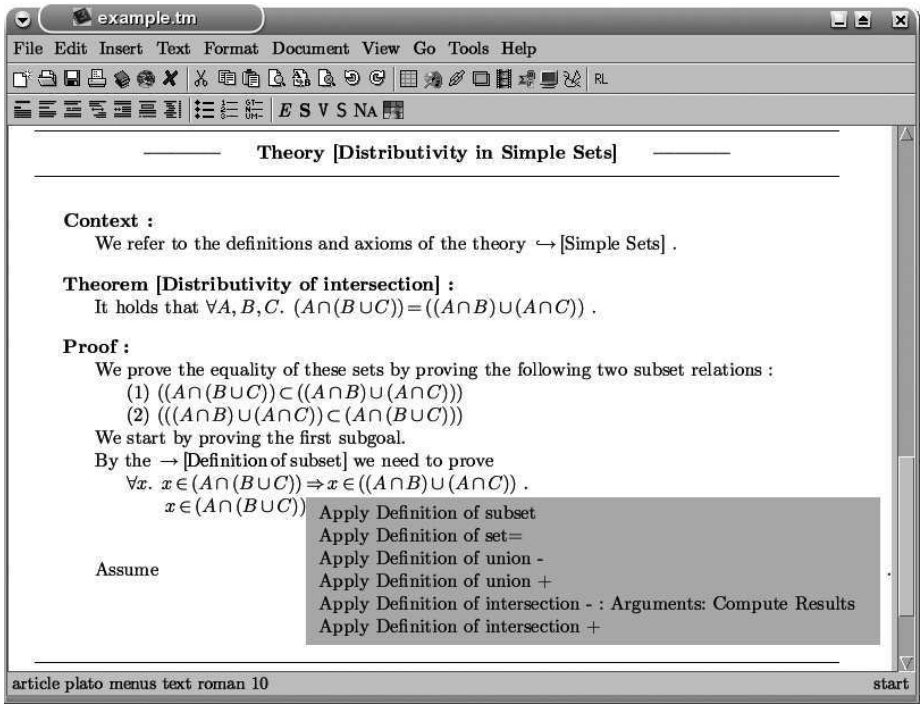


Figure 11. Service Menu in $\TeX_{\text{MACS}}$ requested by the User

able to synchronize his informal representation in the text-editor document with the formal representation in the proof assistance system.

The next interesting feature of PLATΩ is the possibility of getting system support from the underlying proof assistance system. Selecting the recently introduced formula in the assumption, the user requests a service menu from PLATΩ.

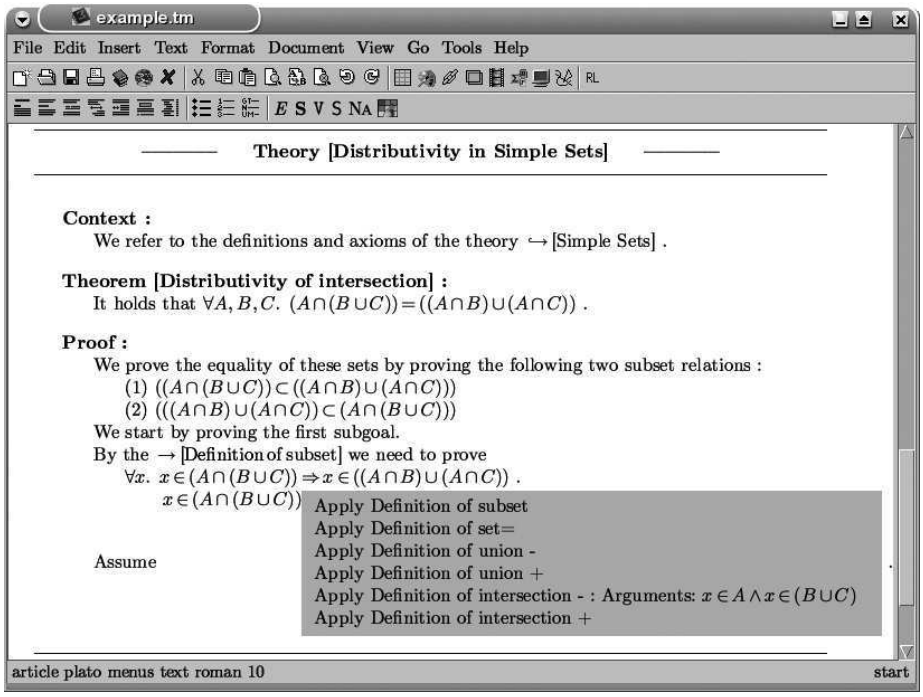


Figure 12. Modification of the Service Menu in TEX_{MACS} by PLATΩ

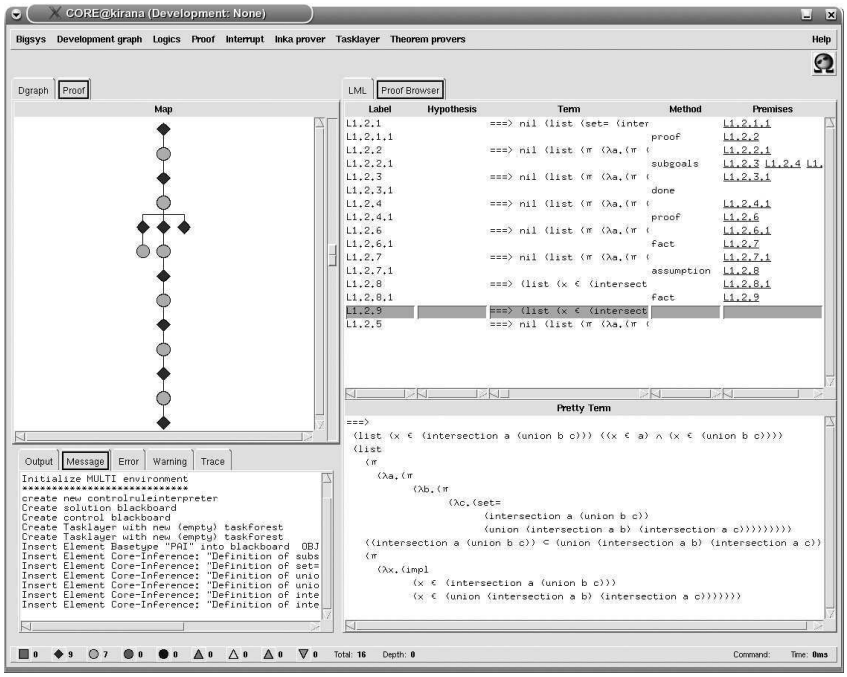


Figure 13. Modification of the Proof in ΩMEGA by the System

Requesting services for the corresponding task in the TASK LAYER, a list of available inferences is returned to PLATΩ. In order to answer quickly to the text-editor, we generate nested actions that allow to incrementally compute the formulas resulting from the application of an inference rather than to precompute all possible resulting formulas for all available inferences. For this example, the inferences were manually generated in the proof

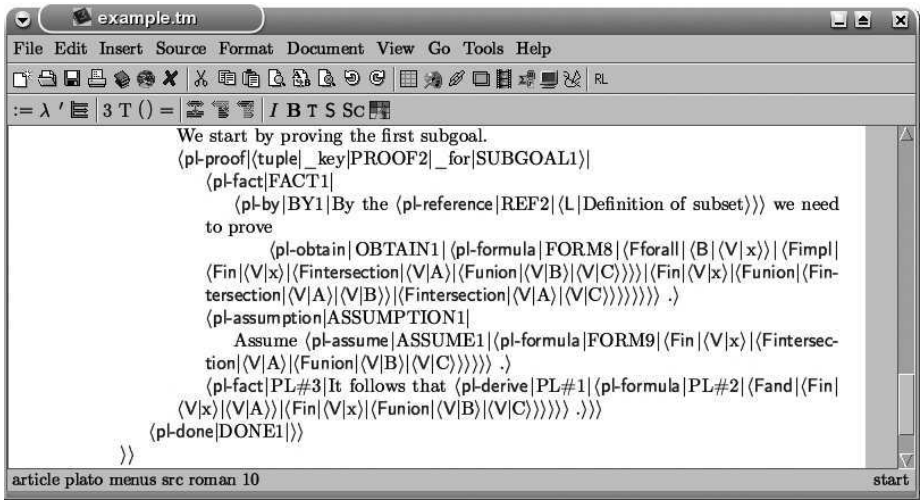


Figure 14. Modification of the Encoding of the Proof in TEXMACS by PLATΩ

assistance system, since the automatic inference generation from the theory knowledge in the development graph is still under development.

The menu is displayed to the user in TEXMACS as shown in Fig. 11, where we already expanded the action **Apply Definition of intersection -** to its nested action **Compute Results**. Executing **Compute Results** calls the method `plato:execute` in PLATΩ, which leads in the TASK LAYER to the computation of all resulting formulas for the inference **Definition of intersection -**, defined by the corresponding axiom. PLATΩ tells the text-editor how to change the menu by sending a patch description for the menu.

The user selects the desired formula (see Fig. 12) which triggers the application of the top level inference and launches a `plato:execute`. PLATΩ calls the TASK LAYER for the application of the selected inference in order to obtain the chosen formula. The TASK LAYER performs the requested operation which typically modifies the proof data structure (see Fig. 13). This modification is transformed by the PLATΩ instance for ΩMEGA into a patch description for the formal representation in TL.

After that, PLATΩ transforms this TL patch into an IL patch and finally a PL patch for the informal document in TEXMACS, which is then send to the text-editor. Furthermore, the menu is closed by sending a patch description which removes it. Currently, the new proof fragments are inserted together with additional predefined natural language fragments. However, we plan to integrate the natural language proof presentation system P.REX [15] into PLATΩ, in order to generate true natural language output for the proof steps added by the proof assistance system.

The text-editor finally patches the encoding of the document (see Fig. 14) according to this patch description. Fig. 15 shows the patched document displayed in TEXMACS.

Note that the user can change any part of the document, including the parts generated by the proof assistance system. Due to the maintenance of consistent versions, the further development of the document can be a mix of manual authoring by the user and interactive authoring with the proof assistance system.

Last but not least, closing the document or the text-editor will close the active session in PLATΩ and in the proof assistance system ΩMEGA by calling the method `plato:close`. For this evaluation we chose a simple mathematical domain in order to focus on the system

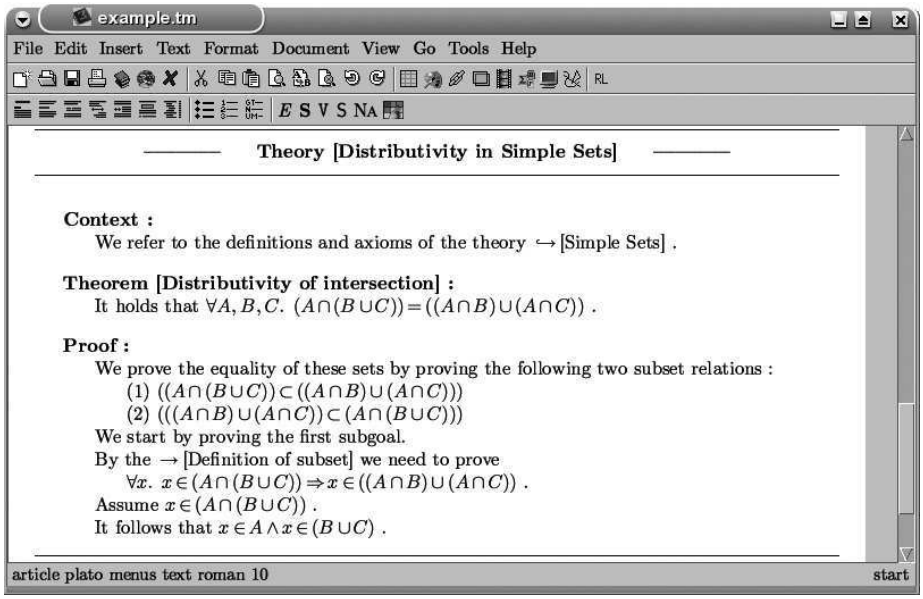


Figure 15. Modification of the Proof in TEX$_{MACS}$ by PLATΩ

behavior of the mediator. In general, the problem solving capabilities only depend on the underlying proof assistance system. Cutting edge proof assistance can be provided by extending the representational sublanguages for complicated domains.

5 Related Work

The AUTOMATH project [13] of Nicolas de Bruijn and his idea of a mathematical vernacular has to be mentioned as pioneering work in the field. Similar to AUTOMATH, the MIZAR [3] and ISAR [29] projects aim at a well balanced compromise between rigorous, formal representation languages suitable for machine processing and human readable, natural representations. The "grammatical framework" approach (GF) [23] goes one step further and employs a λ-calculus based formalism to define grammars consisting of an abstract and a concrete syntax. In the abstract syntax one can formally represent mathematical definitions, theorems and proofs and check their correctness and the concrete syntax defines a mapping from the abstract syntax into linguistic objects. A common problem of these approaches are the diverging requirements of representation to the machine and the user side. AUTOMATH as well as MIZAR and ISAR sacrifice readability to obtain machine processability. GF in contrast shows high readability as well as machine processability but the supported fragment of natural language is far too small and inflexible to allow mathematicians to use their familiar language.

Many mathematical assistance systems favor machine processability over human authoring, while trying to enhance readability. This is done by separating input and output language: the input language remains machine oriented whereas the output language gets close to natural language. The system PCOQ [2] for example uses a schematic approach to represent its output in quasi-natural language. The systems NUPRL [16], CLAM [1] and ΩMEGA/P.REX [15] go further and use natural language processing techniques to generate

[3] www.mizar.org

true natural language output. THEOREMA [12] is a system which strictly separates informal and formal parts in mathematical documents: The user can input informal parts of text without any restriction but these parts are not used for machine processing. The formal parts, however, have to be written in the input language of the computer algebra system MATHEMATICA.

In contrast to that, we suggest in our approach [7] a formal representation language for mathematical content detached from any particular logic or calculus. This allows us to represent arbitrary content regardless of the underlying logic. Moreover, the language allows us to represent both different levels of concept and underspecification and is thus particularly well-suited to represent proofs that are authored in a natural way by human beings. Closely related to our approach is the MATHLANG project [18]. It also proposes a top-down approach starting from natural mathematical texts towards machine processing. However, the MATHLANG project so far concentrates mainly on supporting the analysis of the abstract representations based on type checking and, in contrast to our approach, the gap between real theorem provers and mathematical assistance tools remains open.

To our knowledge there has not been any attempt to integrate a proof assistance system with text-editors in the flexible way as done via PLATΩ. All approaches described above do not consider the input document as an independent, first-class citizen with an internal state that has to be kept consistent with the formal representations in the proof assistance system while allowing arbitrary changes on each side. The only work in that direction has been carried in the context of PROOF GENERAL [3]. In PROOF GENERAL the user edits a central document in a suitable editing environment, from which it can be evaluated by various tools, such as a proof assistant, which checks whether the document contains valid proofs, or a renderer which typesets or renders the document into human oriented documentation readable outwith the system. However, the system is only an interface to proof assistance systems that process their input incrementally. Hence, the documents edited in PROOF GENERAL are processed incrementally in a top-down manner and especially parts that have been processed by the proof assistance systems are locked and cannot be edited by the user. Furthermore, the documents are in the input format of the proof assistant rather than in the format of some type-setting program. Though we have tried to design the functionalities and representation languages in PLATΩ's interface as general as possible, future work will have to show that PLATΩ can be as easily adapted to different proof assistants as is already possible for PROOF GENERAL.

6 Conclusion

The main contribution is the design and development of a generic mediator, called PLATΩ, between text-editors and the proof assistance system ΩMEGA. The presented mediator allows the user to write his mathematical documents in the language he is used to, that is a mixture of natural language and formulas. These documents are semantically annotated preserving the textual structure by using a flexible parameterized proof language. PLATΩ automatically builds up the corresponding formal representation in ΩMEGA and takes further care of the maintenance of consistent versions while providing a mechanism to propagate changes between both representations. All kinds of services of the underlying proof assistance system regarding the formal representation can be provided through PLATΩ by context-sensitive service menus in order to support the interactive development of math-

ematical documents at a high level of abstraction. Altogether, PLATΩ contributes to the evolution of proof assistance systems towards ideal mathematical assistance systems.

In this paper we have illustrated how informal, natural proofs developed in the text-editor are mapped to formal representations in ΩMEGA. Does this mapping already imply that the informal proofs are validated? Clearly not, since ΩMEGA proof sketches at the TASK LAYER may be unsound and only full expansion of these proof sketches to the CORE-calculus layer will assure soundness. In our approach, this expansion can ideally be automated by ΩMEGA's reasoning tools. However, this clearly depends on the structural quality and the granularity of the informal proof. And, of course, if the informal proof is wrong, the expansion will fail and an interaction with the user to patch the proof is required.

References

[1] Marianthi Alexoudi, Claus Zinn, and Alan Bundy. English summaries of mathematical proofs. In Christoph Benzmüller and Wolfgang Windsteiger, editors, *Second International Joint Conference on Automated Reasoning — Workshop on Computer-Supported Mathematical Theory Development*, pages 49–60, University College Cork, Cork, Ireland, 2004.

[2] Ahmed Amerkad, Yves Bertot, and Laurence Rideau. Mathematics and proof presentation in Pcoq. In Uwe Egly, Armin Fiedler, Helmut Horacek, and Stephan Schmitt, editors, *Proceedings of the Workshop on Proof Transformation, Proof Presentations and Complexity of Proofs (PTP-01)*, pages 51–60. Universitá degli studi di Siena, 2001.

[3] David Aspinall, Christoph L¨th, and Burkhart Wolff. Assisted proof document authoring. In Michael Kohlhase, editor, *Mathematical Knowledge Management MKM 2005*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 65–80. Springer, 2006.

[4] S. Autexier. The CORE calculus. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, LNAI 3632, Tallinn, Estonia, july 2005. Springer.

[5] Serge Autexier, Christoph Benzmüller, Dominik Dietrich, Andreas Meier, and Claus-Peter Wirth. A generic modular data structure for proof attempts alternating on ideas and granularity. In Michael Kohlhase, editor, *Proceedings of MKM'05*, volume 3863 of *LNAI*, IUB Bremen, Germany, january 2006. Springer.

[6] Serge Autexier and Dominik Dietrich. Synthesizing proof planning methods and oants agents from mathematical knowledge. In Jon Borwein and Bill Farmer, editors, *Proceedings of MKM'06*. Springer, august 2006.

[7] Serge Autexier and Armin Fiedler. Textbook proofs meet formal logic - the problem of underspecification and granularity. In Michael Kohlhase, editor, *Proceedings of MKM'05*, volume 3863 of *LNAI*, IUB Bremen, Germany, january 2006. Springer.

[8] Serge Autexier and Dieter Hutter. Formal software development in maya. In Dieter Hutter and Werner Stephan, editors, *Festschrift in Honor of J. Siekmann*, volume 2605 of *LNAI*. Springer, february 2005.

[9] Chr. Benzmüller and V. Sorge. ΩANTS – an open approach at combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *Proceedings of Calculemus-2000*, St. Andrews, UK, 6–7 August 2001. AK Peters.

[10] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. Xml path language (xpath) 2.0, 2005. http://www.w3.org/TR/xpath20.

[11] T. Bray, J. Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (xml) 1.0 (third edition), 2004. http://www.w3.org/TR/REC-xml.

[12] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. An Overview of the Theorema Project. In *Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC'97)*, Hawaii, 1997. ACM Press.

[13] Nicolaas G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, pages 29–61, 1970.

[14] Dominik Dietrich. The Task Layer of the ΩMEGA System. Diploma thesis, Saarland University, Saarbrücken, Germany, 2006.

[15] Armin Fiedler. *User-adaptive Proof Explanation*. Phd thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[16] Amanda M. Holland-Minkley, Regina Barzilay, and Robert L. Constable. Verbalization of high-level formal proofs. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99) and Eleventh Innovative Application of Artificial Intelligence Conference (IAAI-99)*, pages 277–284. AAAI Press, 1999.

[17] Xiaorong Huang. *Human Oriented Proof Presentation: A Reconstructive Approach*. Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.

[18] F. Kamareddine, M. Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In M. Kohlhase, editor, *MKM 2005, Fourth International Conference on Mathematical Knowledge Management*, LNAI 3863, pages 217–233. Springer, 2006.

[19] Michael Kohlhase. OMDOC: Towards an Internet standard for the administration, distribution, and teaching of mathematical knowledge. In John A. Campbell and Eugenio Roanes-Lozano, editors, *Proceedings of Artificial intelligence and symbolic computation (AISC-00)*, volume 1930 of *LNCS*. Springer, 2001. http://www.mathweb.org/omdoc.

[20] Andreas Laux and Lars Martin. Xupdate xml update language (working draft), 2000. http://xmldb-org.sourceforge.net/xupdate.

[21] A. Meier and E. Melis. MULTI: A multi-strategy proof-planner. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, LNAI 3632, Tallinn, Estonia, july 2005. Springer.

[22] Svetlana Radzevich. Semantic-based diff, patch and merge for xml-documents. Master thesis, Saarland University, Saarbrücken, Germany, 2006.

[23] Aarne Ranta. Grammatical framework — a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

[24] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, and Martin Pollet. Proof development with OMEGA: $\sqrt{2}$ is irrational. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002*, number 2514 in LNAI, pages 367–387. Springer, 2002.

[25] Jörg Siekmann, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet, and Volker Sorge. LΩUI: Lovely Ωmega user interface. *Formal Aspects of Computing*, 11:326–342, 1999.

[26] V. Sorge. *A Blackboard Architecture for the Integration of Reasoning Techniques into Proof Planning*. PhD thesis, FR 6.2 Informatik, Universität des Saarlandes, Saarbrücken, Germany, Novermber 2001.

[27] Joris van der Hoeven. Gnu $\text{TE}\text{X}_{\text{MACS}}$: A free, structured, wysiwyg and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.

[28] Marc Wagner. Mediation between text-editors and proof assistance systems. Diploma thesis, Saarland University, Saarbrücken, Germany, 2006.

[29] Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: TPHOLs'99*, volume 1690 of *LNCS*, pages 167–184. Springer Verlag, 1999.

[30] F. Wiedijk. Formal proof sketches. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs: Third International Workshop, TYPES 2003*, LNCS 3085, pages 378–393, Torino, Italy, 2004. Springer.

[31] Dave Winer. Xml-rpc specification, 1999. http://www.xmlrpc.com/spec.

Appendix

A PLATΩ's Interfaces

In this section we provide detailed descriptions of PLATΩ's abstract interface functions:

- **Initialize a session**:

  ```
  plato:init (client.name) ->  session.name
  ```

  initializes a new session. It takes the client name (string) as only argument and returns the session name (string) or an error message. The purpose is to start a session in PLATΩ and in the proof assistance system in order to get a session identifier which can be used to indicate the working session in all following interactions with PLATΩ. This is important, for example, if the text editor user wants to get support for two or more documents, or if PLATΩ is launched as mathematical service provider to allow text-editors the access over the web.

- **Upload a document**:

  ```
  plato:upload (session.name, document) ->  OK
  ```

uploads a whole document in the informal language PL. The arguments are the session name (string), received previously by `plato:init`, and the document (string). It returns a simple `OK` (boolean) or an error message. PLATΩ first verifies the syntax of the document and then automatically builds up the corresponding formal representations DL and TL, which are uploaded into the proof assistance system. If a document has already been uploaded, PLATΩ performs an internal difference analysis using a semantic based differencing mechanism [22] and then proceeds as with patching the document.

- **Patch a document**:

  ```
  plato:patch (session.name, diff) ->  OK
  ```

  patches an already uploaded document in the informal language PL with patch information. The arguments are the session name (string) and the patch information (XUPDATE, see Section 3.2). PLATΩ returns a simple `OK` (boolean) or an error message. PLATΩ transforms this patch information into patches for the formal representations DL and TL, which are used to patch the datastructure of the proof assistance system.

- **Request a menu**:

  ```
  plato:service (session.name, object.id) ->  menu
  ```

  requests a menu for an object in the informal language PL inside the document. The arguments are the session name (string) and the unique identifier of the selected object (string). The response is either a menu in the service language SL (string) or an error message. The purpose is to use `plato:service` in order to get a service menu from the proof assistance system with actions for the selected object in the document. PLATΩ looks into his maptable for the corresponding objects in the formal representation and requests service support from the proof assistance system on these objects.

- **Execute a menu action**:

  ```
  plato:execute (session.name, action.id, arguments)
                    ->  (menu.diff, document.diff, custom)
  ```

  triggers the execution of an action with its evaluated arguments. The arguments are the session name (string), the unique identifier of the selected action (string) and the arguments as a list of pairs with name (string) and value (string). It returns a list with a patch for the current menu (string), a patch for the document (string) and a custom answer (string), or an error message.

- **Close a session**:

  ```
  plato:close (session.name) ->  OK
  ```

  closes a session. The argument is the session name (string). It returns a simple `OK` (boolean) or an error message. The purpose is to terminate a session appropriately, such that PLATΩ as well as the proof assistance system are able to delete any information regarding this session.

Tool Support for Proof Engineering

Anne Mulhern[1,2] Charles Fischer[3] Ben Liblit[4]

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI USA

Abstract

Modern integrated development environments (IDEs) provide programmers with a variety of sophisticated tools for program visualization and manipulation. These tools assist the programmer in understanding legacy code and making coordinated changes across large parts of a program. Similar tools incorporated into an integrated proof environment (IPE) would assist proof developers in understanding and manipulating the increasingly larger proofs that are being developed. In this paper we propose some tools and techniques developed for software engineering that we believe would be equally applicable in proof engineering.

Keywords: IDE, IPE, proof visualization, program visualization, refactoring, program extraction, Coq, proof dependencies, proof transformations, proof strategies, proof framework, proof reuse, proof explanation

1 Introduction

Modern integrated development environments (IDEs) provide programmers with a variety of sophisticated tools for program understanding and manipulation. In addition to such basics as syntax highlighting and project building, these tools commonly offer refactorings and program visualization components. Many of the techniques developed for IDEs can be transferred directly to the world of UITPs. Others can be modified to exploit the special nature of theorem provers.

The idea of transferring IDE techniques to theorem provers is not new [2,7,21,36]. However, there have been significant advances in IDEs in the last decade. Many of these advances have been motivated by the needs of developers who must maintain and extend large bodies of existing code. The increasing complexity of real world programs means that even an experienced programmer will struggle to understand the relationships between different software components. When extending or fixing existing code the programmer may spend hours or days merely figuring out what

[1] This work has been supported in part by funding from a Graduate Women in Science Ruth Dickie Grant-in-aid award.

[2] Email: mulhern@cs.wisc.edu

[3] Email: fischer@cs.wisc.edu

[4] Email: liblit@cs.wisc.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

other parts of the program these changes may affect. Moreover, the changes the programmer must make may be scattered across several program components. For this reason, numerous software management tools have been developed to assist in visualizing program properties. Others allow a programmer to navigate a project easily and to make automatic changes across multiple files.

As automated theorem proving matures, the proportion of old proofs to new as well as their size will continue to grow. Tools to visualize, understand, and automatically change these proofs will become vital. Integrated proof environments (IPEs) [5] should incorporate these tools in the same manner as IDEs.

In the following sections we discuss several techniques useful in software development that can be extended to theorem proving. These techniques are navigation by derivation, multiple views, automatic refactorings, and proof visualization in the large.

2 Navigation by Derivation

Formal proofs, even relatively simple ones, are necessarily very large. For example, a formalization of the Sudoku puzzle and an accompanying solution procedure in Coq [35] required approximately 5000 lines. A formal proof of the four color theorem [12, 41] took about 60,000 lines and a few years to develop. Sophisticated automated proof assistants have been developed to assist in the construction of such proofs using *tactics*. These tactics may be manually selected by the user or automatically chosen by the proof assistant. The structure of a proof object generated by these tactics may be difficult for a user to predict even when the user has selected the tactic. When a tactic is selected automatically the structure may be further obscured. The proof objects themselves may be far too large to be easily read. For example, the Sudoku development mentioned above contains a proof that the permutation relation on two lists is invertible. That is, where a pair of lists are permutations of each other, and the head elements of the lists are equal, the tails of the two lists must also be permutations of each other. About ten lines of tactics are required to complete the proof of the theorem, but at roughly 750 lines the generated proof is two orders of magnitude larger. Nonetheless, there are many occasions on which it becomes necessary to study such proofs. A tactic implemented in a proof assistant may not be working as expected; it may be necessary to inspect proof objects themselves in order to debug the tactic. A user may be developing a proof specifically to exploit a proof assistant's extraction mechanism and may need to inspect the proofs to understand why the extracted code is inefficient or, in some cases, non-existent [8]. It may be necessary to rediscover what auxiliary theorems were used to prove a given theorem; such auxiliary theorems may be selected without the user's intervention by a proof assistant with support for automation.

Most programmers are familiar with the Unix *diff* utility which identifies the textual differences between two files. A number of visual tools exploit an underlying diff tool. For example, the Eclipse Compare view allows the user to compare up to three files. The tool automatically aligns the differences between the files and

[5] The authors would like to thank one of the anonymous reviewers for acquainting them with this term.

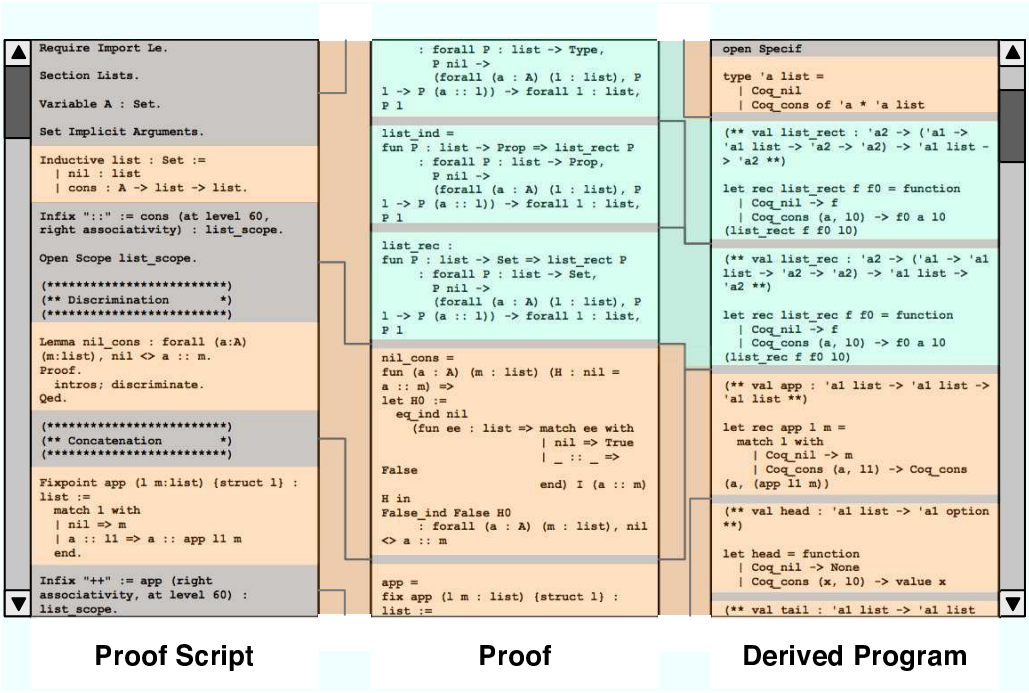


Fig. 1. Overall structure of a three panel proof navigation tool. The proof is taken from the Coq List library, one of the standard libraries in the Coq distribution. The scroll bars on the left and right allow the user to navigate the proof script and the derived program respectively.

matches corresponding parts using visual cues. This technique, using visual cues to identify associated entities, can be extended to other domains. For example, a proof developer will often have two perspectives on a given proof. The first perspective consists of the definitions and theorems along with their corresponding tactics. The second perspective consists of the same definitions and theorems, this time associated with their proofs. There is a correspondence between the tactics and the terms of the proof. This correspondence differs from that arising in file comparison. In one way it is more straightforward since the proof has a formal relationship to the tactics whereas in a file comparison the relationship between the files must be discovered by an heuristic. However, the correspondence is also more complex. One tactic may correspond to multiple terms in a proof. Hence, an interactive tool which allows the user to select a tactic or group of tactics and responds by highlighting the associated terms in a proof would be a valuable aid to proof understanding.

A number of theorem provers, e.g., PX [13], Minlog [22], Isabelle/HOL [23], NuPRL [24] and Coq [34], exploit the Curry-Howard isomorphism [10,40] to offer a *program extraction* facility [19,20,27]. A program extraction facility automatically generates programs from proofs. In the extraction process the logical parts of a proof are deleted and the computational parts are translated into the source code of the target language. Programs extracted from the proofs of their desired properties are known as *certified* programs. As long as the extraction facility and proof checker are themselves correct, a certified program is guaranteed to be a correct implementation of its specification, i.e., the proof from which it is extracted. Generally, the extracted programs are several orders of magnitude smaller than their associated proofs and

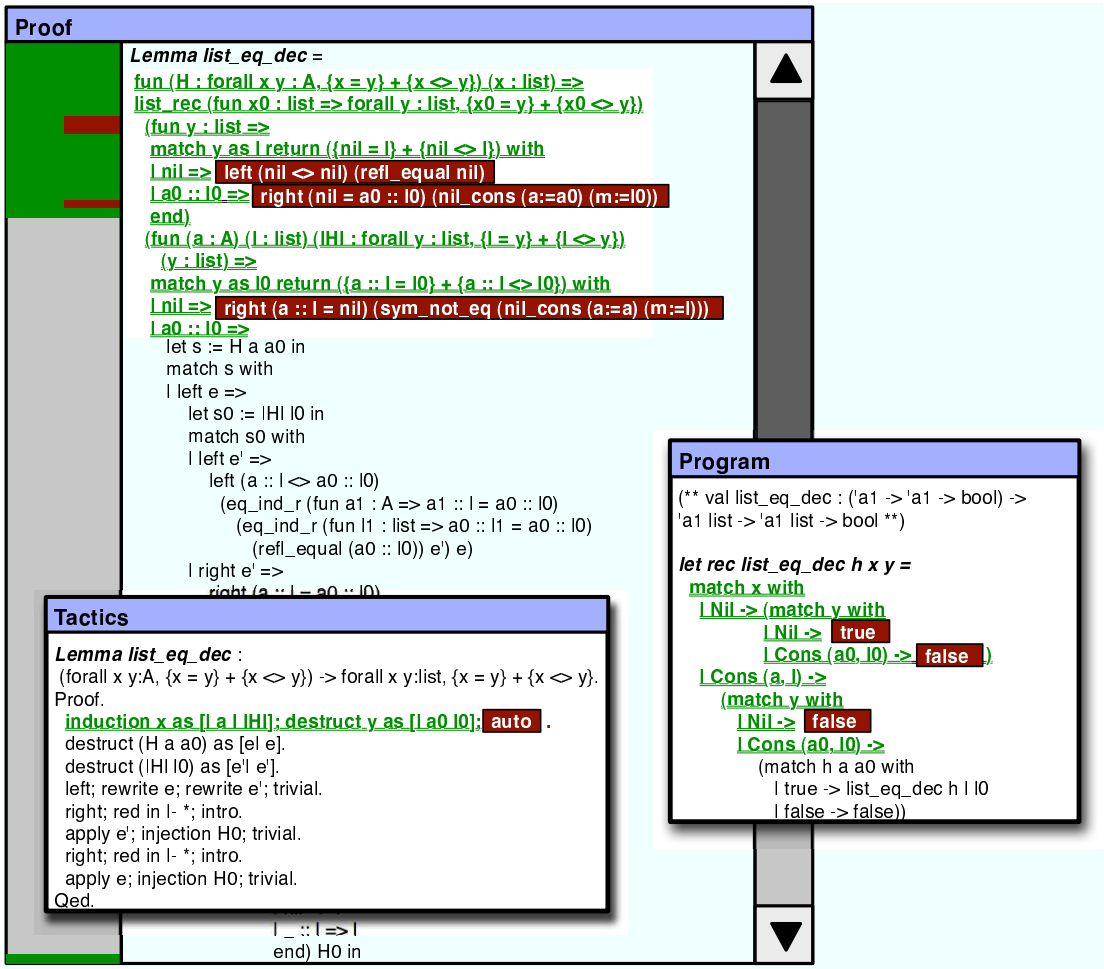


Fig. 2. Proof of the decidability of equality on lists. The *Tactics* pane on the left displays the proof tactics while the *Program* pane on the right displays the extracted program. The *Proof* pane displays the proof proper.

much easier to understand. In the case of theorem provers with an extraction mechanism a three way association would be appropriate and useful. Figure 1 shows the overall structure of such a navigation tool.

Each component is associated with its corresponding component in the adjacent panel. Examples of proof script components are definitions or theorems with tactics, examples of proof components are definitions or proofs, examples of components in an extracted program are definitions of types or functions. Corresponding components are automatically aligned as the user focuses on different areas in the proof script or extracted program. Light gray is used for portions of the proof script that are not incorporated into the proof such as directives to the proof engine or comments. Narrow gray bars are also used to separate proof and program components. Pale blue indicates that a component has been generated indirectly from a component in the proof script. In this example, some induction principles for the list type have been automatically generated. Some components of the proof do not have corresponding components in the extracted program. In this case the adjacent separators are merged in the program pane.

The tool in Figure 1 is useful for high-level inspection. The user may also want

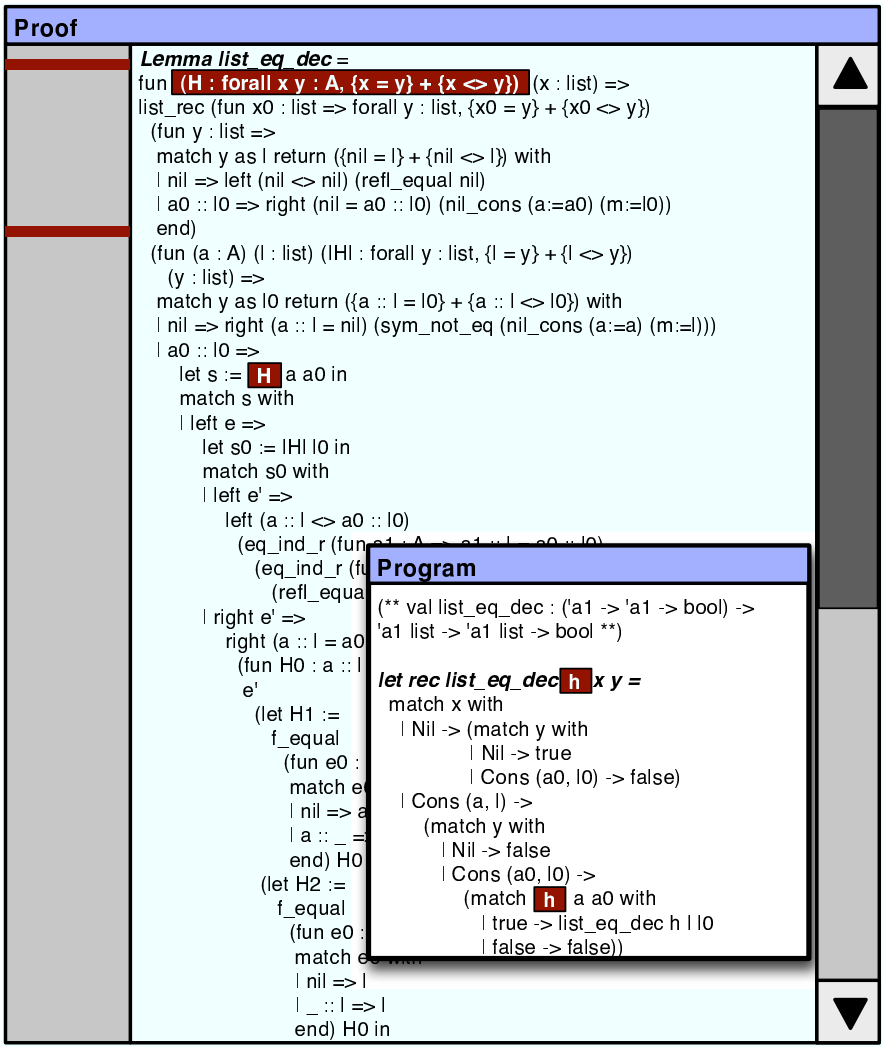


Fig. 3. Proof of the decidability of equality on lists. The user has highlighted the `h` parameter in the `list_eq_dec` function. Uses of the `h` parameter in the function and the corresponding `H` parameter of the proof are highlighted.

to examine individual proof entities in more detail. Figure 2 shows a proof and its associated tactics and program. In the *Tactics* pane on the left the `auto` tactic has been selected. Preceding tactics are green and subsequent tactics are left in black. The proof terms generated by the highlighted tactic are themselves highlighted and proof terms generated by the preceding tactics are in green. The bar on the left of the *Proof* pane summarizes the entire proof. Note that there is a green line at the bottom of the bar indicating that the last few lines of the proof are generated by the tactics preceding `auto`. The *Program* pane on the right shows the extracted program. The corresponding terms in the generated program are highlighted.

In the preceding example, elements in the proof were selected via the proof script. It is also possible to select these elements via the extracted program or to select elements in the program via the proof. Figure 3 shows the same proof as before. In this example, however, the user has selected an element in the *Program* pane, specifically `h`, the formal argument of the `list_eq_dec` function. Uses of `h` in `list_eq_dec` and corresponding elements in the proof are highlighted. The summary

```
Require Import Le.

Section Lists.

Variable A : Set.

Set Implicit Arguments.

Inductive list : Set :=
  | nil : list
  | cons : A -> list -> list.

Infix "::" := cons (at level 60, right associativity) : list_scope.

Open Scope list_scope.

(*************************)
(** Discrimination       *)
(*************************)

Lemma nil_cons: forall (a:A) (m:list), nil <> a :: m.
Proof.
  intros; discriminate.
Qed.
```

Fig. 4. View of a proof script showing syntax highlighting. The highlighting scheme is adapted from that in the CoqIDE.

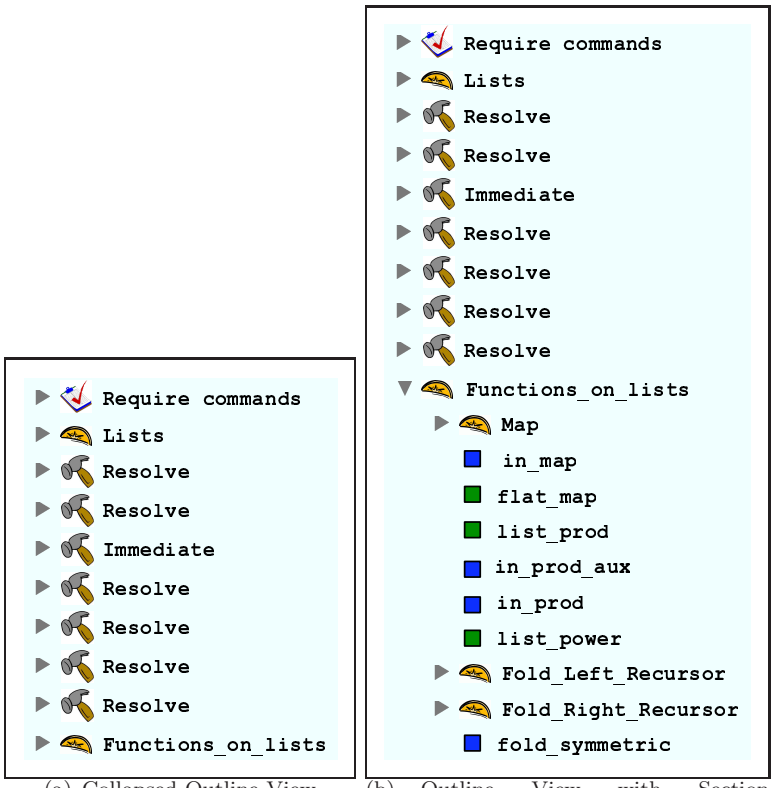
bar in the *Proof* pane indicates that there are no matches other than those visible in the text. This confirms our intuition about the proof. `h` is a function which decides whether two list elements are equal. Its corresponding proof, `H`, is a proof of the decidability of equality on list elements. `h` is applied to the head element of each list to determine whether the two are equal and in the case where the elements are equal is passed as an argument in the recursive call (otherwise `list_eq_dec` returns false). In the corresponding inductive proof we would expect that `H` is also used just once, as an hypothesis in the proof that lists are equal if their heads and their tails are equal, and we see that this is the case.

When a program is compiled with debugging enabled the compiler encodes extra information for the debugger's use in the generated object files. In particular, it stores debugger "symbol tables" [33] which are mappings between the source code and the generated object code. Using this information a symbolic debugger can execute a machine instruction and yet display to the user the corresponding source code. We envision a similar approach for a theorem prover. As the prover executes tactics to generate a proof it can store a mapping between the tactics and the generated proof object, making it available to a program navigation tool such as that described above. We have observed that the correspondence between the tactics and the proof object may be complex; but compilers and debuggers are able to generate and navigate the equally intricate mappings between source code and highly optimized machine code.

3 Common Conveniences

3.1 Multiple Views

Syntax highlighting, which is ubiquitous in IDEs, is available in some form in a number of proof assistants [29,34]. Figure 4 shows a Coq proof script. The various sorts of keywords are distinguished by the use of different colors, and this helps us to



(a) Collapsed Outline View (b) Outline View with Section `Functions_on_lists` expanded

Fig. 5. View of a proof script outline.

understand the basic structure of the small portion of the program we are looking at. When we zoom out, the syntax coloring becomes virtually useless. But this problem can be addressed by techniques already in use in a number of IDEs. For example, the Eclipse [9] Java Perspective provides an *Outline* view which allows the user to see the basic structure of an individual file at a glance. The *Outline* view is used for navigation as well. Figure 5 shows a suggested outline for the proof script of Figure 4. Another idea that could be extended directly to proof assistants is the technique of collapsing and expanding parts of a source file. Often a programmer wishes to elide certain parts of a source file that are irrelevant, so that the rest of the file becomes easier to understand. In a similar fashion a proof developer may wish to elide portions of a proof script, of a proof, or of its associated program. Figure 6 shows the proof of the decidability of equality on lists with two of the functions in the proof collapsed. The first collapsed function is a proof that equality of the heads of the lists is irrelevant under the hypothesis that the tails are unequal (in which case it is clear that the lists are unequal). The second function is a similar proof, with heads and tails reversed. Such subproofs, although required to complete a formal proof, and in some cases constituting a significant proportion of the whole proof, are generally uninteresting to the human reader.

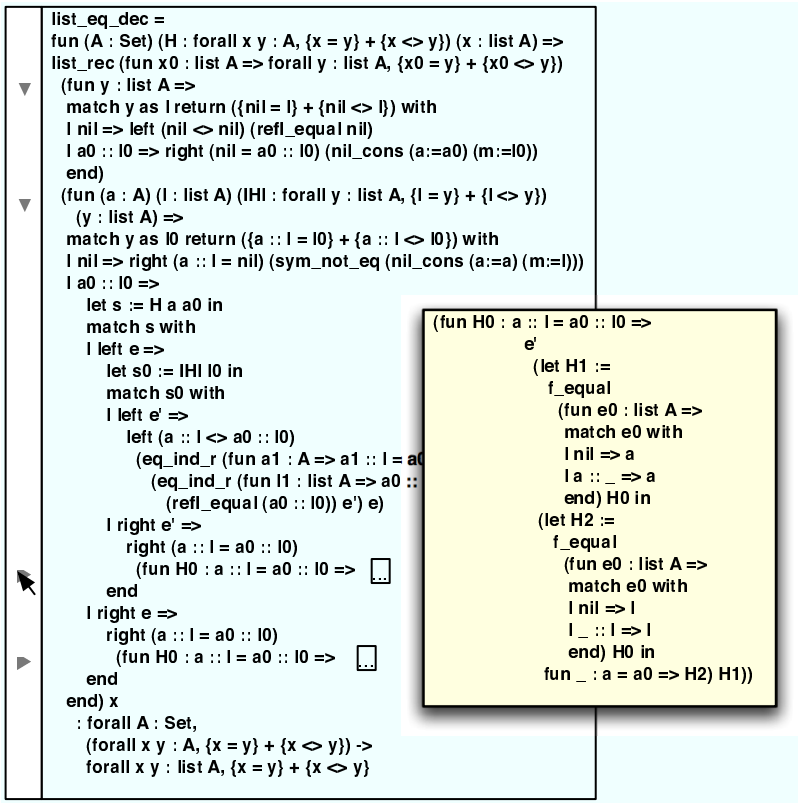


Fig. 6. A proof of the decidability of equality on lists with two functions collapsed. The collapsed function is inspected by allowing the cursor to hover over the arrow; pressing the arrow causes the function to be expanded.

3.2 Automatic Refactoring

A refactoring is a way of restructuring a program so that the overall organization of the program is improved but the behavior is unchanged [25]. Where large parts of a proof have been developed separately, refactoring may be necessary to make common the underlying assumptions of the different components [12]. Refactorings may also facilitate proof reuse [16]. While modern IDEs offer extensive support for automatic refactorings [30, 37, 38] UITPs offer very little. IDEs offer support for renaming of functions and variables; UITPs should offer a similar facility for renaming lemmas. IDEs offer facilities for restructuring programs; for example, a local variable may be converted to a field in a Java class definition. In the same way, UITPs should offer facilities for restructuring existing proof scripts; in Coq, for example, a user might wish to encapsulate a group of proof entities within a module. In the Eclipse Java IDE, a developer can generalize the type of a field, lifting the field to its supertype [38] and changing all uses of the field appropriately. Similarly, UITPs should offer refactoring support for abstracting over definitions and hypotheses [28]. Many other refactorings are likely to be dependent on the logic and organization of the individual proof assistant.

Additionally we propose a requirement for transformations similar to the "best effort" standard used by IDE developers. When a developer changes the signature of a method an IDE may "do its best" by changing the signature of all overriding and overridden methods appropriately. However, if the signature is changed by

the addition of a formal parameter, it will generally be impossible to automatically determine the actual parameter to be passed at the invocation site. After the transformation the resulting type mismatch will induce compiler errors in the program. However, the IDE has eased the programmer's task by automatically performing a task that the programmer would otherwise need to perform manually. The programmer can complete the transformation by identifying the call sites that must be changed, determining the actual parameter to be passed at each call site, and updating the code correctly. Generally, the compiler itself will assist the programmer in identifying the call sites which must be updated through specific error messages.

UITP developers may feel that an automatic transformation that makes a correct proof incorrect is simply unacceptable. We argue that if the transformation gets the proof developer "closer" to the correct proof that he actually desires such a "best effort" transformation is still of value and worth incorporating in a UITP. A developer may realize only after substantial work has been done on a proof that some component must be changed. For example, it may turn out to be the case that a list must have not only the familiar properties of lists but also the extra property that its elements are sorted for a proof to be completed. One method of expressing this additional property in Coq is through the use of dependent types [3]. If the developer changes the type of the list to include a proof that it is sorted then any previously developed theorems that include this list must also have their type changed. It is relatively easy to implements such a straightforward transformation. It may even be possible for a refactoring tool to modify the tactic scripts for certain proofs that do not rely on the sorted property so that the proof can be reconstructed entirely. But perhaps the developer must now construct additional lemmas to prove that the sorted property is preserved by some transformations defined in the proof. The proof cannot be completed without this additional manual work on the part of the developer. Still, a refactoring tool that automated the straightforward steps and left the developer to perform the more difficult steps that cannot easily be automated would be desirable.

4 Proof Visualization in the Large

Program visualization is a well established field. Techniques to represent programs visually are used in teaching [5, 15] and in the professional world [39] and new techniques are continually developed [18, 26, 31, 32]. These techniques incorporate both static visualization [18, 5, 39, 26, 32] and animations [15]. Often they use a complicated visual vocabulary to communicate relationships among many entities in a program.

An important insight of Ball and Eick [1] is that a less complicated visual vocabulary can also convey useful information. They show how a coloring scheme can be used to convey to the programmer the overall "shape" of an application. They use color to encode unary properties of individual lines such as the number of times a line has been changed. Such coloring can allow a programmer to see at a glance some overall property of the program. For example, parts of the system that are predominantly red are edited frequently and most likely contain bugs. Parts that are blue are edited less frequently and are likely to be relatively bug free. This

approach can be extended to textual units of larger granularity such as procedures or files and has been used in applications such as fault localization [17].

Techniques for *proof* visualization are less common. Proof animations [14] exist for restricted domains such as graph properties [11]. Static visualization techniques are used to describe the relationships among proof entities [4,6]. We argue that the insights of Ball and Eick can be applied to proof visualization as well as program visualization. They can be applied in a straightforward way to encode such properties as revision information which are really identical between proofs and programs. Other properties are more specific to UITPs. In a proof assistant with an automatic component theorems may be applied without a user specifically requesting them. A coloring scheme that encoded the relative frequency with which different theorems were used could be used to visualize "hot spots" in much the same way a coloring scheme that encodes software profiling information is used.

5 Conclusion

We have described a number of ways in which techniques developed to assist programmers in maintaining and extending large programs can be of use to proof developers who must maintain and extend large proofs. Many software projects involve a considerable number of people working over several years. As the discipline of automated theorem proving matures proofs of similar size and complexity, which are now considered extraordinary [41], will grow more common. Program extraction is gaining acceptance as a technique for developing programs which must be correct. As these trends continue, the tools we have described will become more and more valuable to proof developers.

Moreover, we feel that the theoretical difficulties of developing the tools that we have described are negligible. For example, the navigation tool described in Section 2 requires an underlying encoding which records the correspondence between the proof script, its associated proof, and the derived program. It is clear that this data is available. The relationship between the entities in a proof script and its corresponding proof must be calculated by the proof engine that develops the proof. Similarly, the relationship of the terms in a proof to the corresponding terms in the extracted program must be calculated by the program extraction mechanism. The difficulty does not lie in establishing these relationships but rather in recording them and displaying them in a useful manner.

On the other hand, work in this area may yield significant theoretical insights. The refactorings described in Section 3.2 are all quite straightforward; just a bit more sophisticated than textual replacement. Some program refactorings are much more ambitious. For instance, Tip et al. [37] describe a refactoring from Java programs that do not exploit a polymorphic type system to ones that do. More ambitious refactorings for theorem provers could very well yield unexpected insights.

Acknowledgements

We would like to thank Dr. Kenneth Kunen for his advice and encouragement.

References

[1] Ball, T. and S. G. Eick, *Software visualization in the large*, IEEE Computer **29** (1996), pp. 33–43.

[2] Bertot, Y., *The CtCoq system: design and architecture*, Formal Asp. Comput. **11** (1999), pp. 225–243.

[3] Bertot, Y. and P. Casteran, "Interactive Theorem Proving and Program Development : Coq'Art: The Calculus of Inductive Constructions," Texts in Theoretical Computer Science **XXV**, Springer, 2004.

[4] Bertot, Y., O. Pons and L. Pottier, *Dependency Graphs for Interactive Theorem Provers*, Technical Report RR-4052, INRIA (2000).

[5] *BlueJ — Teaching Java — Learning Java*, http://www.bluej.org/.

[6] Boite, O., *Proof reuse with extended inductive types*, in: K. Slind, A. Bunker and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science **3223** (2004), pp. 50–65.

[7] Caplan, J. E. and M. T. Harandi, *A logical framework for software proof reuse*, in: *SSR '95: Proceedings of the 1995 Symposium on Software reusability* (1995), pp. 106–113.

[8] Cruz-Filipe, L. and P. Letouzey, *A large-scale experiment in executing extracted programs*, in: *12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus'2005*, 2005, To appear.

[9] *Eclipse.org home*, http://www.eclipse.org/.

[10] Girard, J.-Y., P. Taylor and Y. Lafont, "Proofs and Types," Cambridge University Press, 1989.

[11] Gloor, P. A., D. B. Johnson, F. Makedon and P. Metaxas, *A Visualization System for Correctness Proofs of Graph Algorithms*, Technical Report PCS-TR92-180, Dartmouth College, Computer Science, Hanover, NH (1992).

[12] Gonthier, G., *A computer-checked proof of the Four Color Theorem*.

[13] Hayashi, S. and H. Nakano, "PX: A Computational Logic," MIT Press, Cambridge, MA, USA, 1988.

[14] Hayashi, S., R. Sumitomo and K. Shii, *Towards the animation of proofs—testing proofs by examples*, Theor. Comput. Sci. **272** (2002), pp. 177–195.

[15] *Jeliot :: Home*, http://www.cs.joensuu.fi/~jeliot/.

[16] Johnsen, E. B. and C. Lüth, *Theorem reuse by proof term transformation.*, in: K. Slind, A. Bunker and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science **3223** (2004), pp. 152–167.

[17] Jones, J. A., M. J. Harrold and J. Stasko, *Visualization of test information to assist fault localization*, in: *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (2002), pp. 467–477.

[18] Jones, J. A., A. Orso and M. J. Harrold, *Gammatella: visualizing program-execution data for deployed software*, Information Visualization **3** (2004), pp. 173–188.

[19] Letouzey, P., *A New Extraction for Coq*, in: H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, Lecture Notes in Computer Science **2646** (2003).

[20] Letouzey, P., "Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq," Ph.D. thesis, Université Paris-Sud (2004).

[21] Luo, Z., *Developing reuse technology in proof engineering*, in: *Proceedings of AISB95, Workshop on Automated Reasoning: bridging the gap between theory and practice,*, Sheffield, U.K., 1995.

[22] *Minlog system*, http://www.minlog-system.de/.

[23] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," Number 2283 in Lecture Notes in Computer Science, Springer, 2002.

[24] *PRL Automated Reasoning Project at Cornell*, http://www.cs.cornell.edu/Info/Projects/NuPrl/.

[25] Opdyke, W. F., "Refactoring object-oriented frameworks," Ph.D. thesis, Champaign, IL, USA (1992).

[26] Panas, T., R. Lincke and W. Löwe, *The VizzAnalyzer handbook*, Technical report, Växjö University (2005).

[27] Paulin-Mohring, C. and B. Werner, *Synthesis of ML programs in the system Coq*, J. Symb. Comput. **15** (1993), pp. 607–640.

[28] Pons, O., *Proof generalization and proof reuse*.

[29] *Proof General*, http://proofgeneral.inf.ed.ac.uk/.

[30] Rajesh, J. and D. Janakiram, *Jiad: a tool to infer design patterns in refactoring*, in: *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2004), pp. 227–237.

[31] Rodrigues, N., *Haskell slicing*, http://labdotnet.di.uminho.pt/HaSlicing/HaSlicing.aspx.

[32] Rodrigues, N. F. and L. S. Barbosa, *Component identification through program slicing*, in: *Proceedings of the Second International Workshop on Formal Aspects of Component Software*, 2005.

[33] Rosenberg, J. B., "How Debuggers Work: Algorithms, Data Structures, and Architecture," John Wiley & Sons, Inc., New York, NY, USA, 1996.

[34] The Coq Development Team, "The Coq Proof Assistant Reference Manual," (2004).

[35] Théry, L., *Sudoku in Coq* (2006).

[36] Théry, L., Y. Bertot and G. Kahn, *Real theorem provers deserve real user-interfaces*, in: *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments* (1992), pp. 120–129.

[37] Tip, F., R. Fuhrer, J. Dolby and A. Kieżun, *Refactoring techniques for migrating applications to generic Java container classes*, IBM Research Report RC 23238, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA (2004).

[38] Tip, F., A. Kieżun and D. Bäumer, *Refactoring for generalization using type constraints*, in: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, CA, USA, 2003, pp. 13–26.

[39] *Object Management Group — UML*, http://www.uml.org/.

[40] Wadler, P., *Proofs are programs* (2000).

[41] Wiedijk, F., *The seventeen provers of the world* (2005).

Presenting and Explaining Mizar

Josef Urban[1]

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

Grzegorz Bancerek[2]

Faculty of Computer Science
Bialystok Technical University
ul. Wiejska 45A, Bialystok, Poland

Abstract

The Mizar proof language has both many human-friendly presentation features, and also firm semantical level allowing rigorous proof checking. Both the presentation features and the semantics are important for users, and an ideal Mizar presentation should be both human-friendly (i.e. very close to textbook presentations), and also allowing fast access to the detailed semantics and detailed proof explanations. This poses several questions, problems and choices when presenting original Mizar texts, presenting results of semantic queries over the Mizar library, and also when presenting texts produced directly on the semantical level, e.g. by automated theorem provers. This paper discusses solutions to these problems, and particularly implements an initial system for presenting detailed explanations of atomic Mizar inferences. This is done by the cooperation of the Mizar XML presentation tools, the MML Query system, and automated theorem provers working on the MPTP semantic translation of Mizar.

1 Introduction

One of the main objectives in the development of the Mizar [Rud92,RT99] proof language has always been its intuitive presentation and closeness to mathematical vernacular. The following features are worth mentioning in this context:

- It uses Jaskowski's natural deduction [Pel99,Jas34] for the high-level proof structure, complemented with "simple justification" ("by") steps, which are the atomic inferences checked by the fast Mizar refutational checker [Wie00,NB04]. These atomic steps are fine-tuned to be of the "right" human-like granularity, i.e., they should be easy to understand to humans, but should not bother the reader with too much obvious details.

[1] Email: urban@kti.ms.mff.cuni.cz
[2] Email: bancerek@mizar.org

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- The language is essentially first-order predicate theory, but it supports a number of linguistic features that make in more human-like. This includes, e.g., usage of adjectives and types and implicit usage of their hierarchies and dependencies (called "registrations" or "clusters" for adjectives), implicit usage of various properties (symmetry, reflexivity, projectivity, etc.), Mizar structures, implicit definitional expansions, etc.
- The language supports wide variety of notations and several kinds of symbol overloading, to allow faithful notation for different mathematical fields encoded in the large Mizar Mathematical Library (MML).

On the other hand, the main purpose of having formal proof languages is their mechanized proof checking. This means that all the above mentioned presentation features ultimately have to be transformed to a proof-checkable level with clear semantics. In Mizar, this is done in several compiler-like passes, which gradually transform the syntactic features to their semantic counterparts (possibly informing users about syntactic errors, etc.), and finally check on the semantic level the correctness of the proofs.

1.1 The semantic level of Mizar

The Mizar semantic level is characterized mainly by two transformations

- Formulas are transformed to the Mizar normal form (MNF), which uses only certain logical connectives ($\wedge$, $\neg$, $\top$, and $\forall$).[3]
- The disambiguation of all the notation (symbols and their patterns) into the "constructors". While the former are usually quite complicated and overloaded, constructors are the unique semantical elements (functors, predicates, etc.).

Both these transformations are many-to-one, and in some sense also many-to-many. Multiple user-level formulas can have the same MNF, and multiple user-level notations can end up being expressed in the same way on the constructor level. As for the many-to-many property, it is theoretically possible to have multiple MNF for one user-level formula, but in practice this does not happen, since the Mizar transformation algorithm is deterministic.[4] It is much more possible to have one user notation (symbols and their patterns) transformed to different constructors, since this heavily depends on the Mizar *environment* (e.g. type rules contributing to different ways of the overloading disambiguation).[5] The important consequence of this is that given a piece of a semantic-level Mizar text, there are usually multiple ways how it can be presented.

This semantic level directly serves for a number of purposes: It is used by Mizar itself for the proof checking and for storing the Mizar internal database. It is also used in the MML Query [BR03] searching and presentation system. It also serves as the basis for the formats used in the MoMM [Urb06a] system, the Mizar Proof Advisor and MPTP [Urb04,Urb06b] systems, and for the format used for semantic

[3] The term "semantic correlate" introduced by Roman Suszko is usually used in the Mizar world for MNF.
[4] So if we *defined* MNF as the product of the Mizar transformation algorithm, it would indeed be just many-to-one.
[5] And again, this is just many-to-one, if we fix the particular environment.

browsing in the MizarMode [Urb05,BU04].

It should be noted that this semantic level still expresses the *Mizar logic*, not the standard untyped first-order predicate logic used in current automated theorem provers (ATPs) like E [Sch02], Vampire [RV02], SPASS [Wei01,WBH+02], Otter [McC94] or Prover9. Further processing is needed when that logic is transformed to standard predicate logic: e.g., the Mizar types need to be encoded, all knowledge used implicitly by Mizar (e.g. type hierarchies) has to be expressed explicitly, etc. This is now done in a certain way (characterized mainly by encoding types as predicates) by the MPTP system, however there are also many possible choices in this transformation. Conversely, this transformation is again generally many-to-many, there will usually be multiple ways of encoding pure predicate logic in the Mizar logic.

1.2 Using the semantic level for linked presentation

Recently, the Mizar semantic level has been completely XML-ized [Urb06c], and XSLT tools [6] are being developed for creating linked HTML presentation of Mizar [7] from it. The XML-ized semantic format has been designed so that it is relatively easy to do the HTML linking of symbols and other Mizar resources, and it has been modified several times (usually by adding additional information as XML attributes) using the HTML presentation bottlenecks as a feedback. It currently allows quite faithful re-creation of the original Mizar presentation (see Section 3 for more details), while it also reveals a lot of information computed by the Mizar system (e.g. various formulas computed implicitly - for stating Mizar properties, correctness conditions, etc.), which are normally not accessible to Mizar authors. The main point of using pure XSLT for creating the HTML presentation is that all major browsers today support the XSLT language. This means that Mizar authors can now load the XML file (a by-product of the Mizar verification) directly into their browser whenever they need it during the authoring, and thus immediately get all the additional information contained there.

The XML-ized form of a Mizar article (and hence also the HTML presentation) however does not contain any explanation of the atomic "simple justification" ("by") steps. This kind of explanation was never needed for any purpose for the Mizar processing itself, and its addition (i.e., providing documentation mode for the proof checking of the "simple justification" steps) would involve a very large change of Mizar itself. This means that the users so far could not find out why a particular atomic step was accepted by Mizar. As mentioned above, these steps are designed to be "easy to understand but not unnecessarily verbose" for humans, which is however a very subjective matter depending on many factors. As with the normal natural-language proofs, sometimes the number of "obvious" facts used in an atomic Mizar step can be simply too high for a reader, making the "understanding search space" too large. Humans are also good at occasionally forgetting what should be obvious. One of Mizar's probably greatest contributions to the field of formalization of mathematics is its stress on the readability of proofs (i.e., unlike in the tactical

[6] http://kti.ms.mff.cuni.cz/cgi-bin/viewcvs.cgi/xsl4mizar/miz.xsltxt?view=markup
[7] http://merak.pb.bialystok.pl/mml/

provers, the language is not supposed to be "write-only"). While the readability is a very worthy goal per se (well, why shouldn't all of mathematics be presented in a readable, yet formally correct and mechanically checked way?), this feature of the language actually seems to be quite important in the maintenance of such a large repository of formal mathematics as is MML today, and for its refactoring (e.g., generalizing, reformulating of entire theories, etc.). For all these reasons, providing an optional finer explanation level, which helps to understand the more difficult steps when necessary, should be useful.

1.3 The rest of this paper

We describe our initial solution to the problem of providing and presenting the explanations of the Mizar atomic "simple justification" inference steps. This solution (cf. Diagram 1) uses the ATP technology (now the E-PROVER) for providing the actual explanations as ATP proof objects. The MPTP system is used to transform the Mizar "simple justification" inference steps to ATP problems, and the MML Query system is used to transform the ATP proof objects back into the Mizar notation. The proof objects transformed by MML Query are then linked to the appropriate places in the HTML presentation of Mizar articles, so that users can easily access them, when a particular atomic inference steps is not clear to them.

This kind of processing requires several of the above mentioned many-to-many transformations (mainly in the opposite order than mentioned above). We explain the general algorithm used by MML Query for presenting arbitrary semantic-level formulas in the user-friendly notation. This algorithm has been generally used for presenting the MML Query search results, and we are now using it also for the presentation of the text created by ATPs directly on the semantic level. The MML Query solution to this presentational problem is compared to the solution implemented in the HTML presentation of Mizar articles, and their suitability for different purposes is discussed.

2 Explaining and Presenting Mizar Simple Justifications

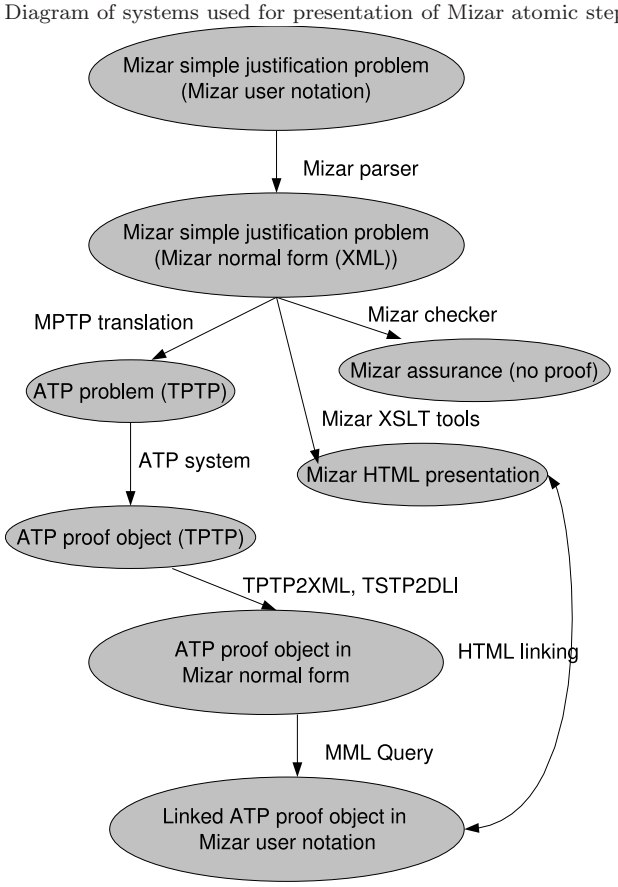
Readers can check the functionality for explaining the Mizar atomic inferences (implemented now for 35 initial Mizar articles) at the authors' web site [8]. This is a development version of the Mizar HTML presentation, very similar to the official one at the Mizar site [9]. The main difference at the moment is the linking of the **by** keyword, which leads to the MML Query rendered ATP proof objects, also available at our site [10]. We provide a simple example below.

2.1 Simple example from user's perspective

Consider e.g. the first Theorem [11] in the Mizar article `ZFMISC_1` [Byl89]:

[8] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/
[9] http://mmlquery.mizar.org/mml/4.48.930/
[10] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/_by/
[11] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/zfmisc_1.html#T1

Fig. 1. Diagram of systems used for presentation of Mizar atomic steps




```
theorem Th1: :: ZFMISC_1:1
  bool {} = {{} }
proof
  now
    let c1 be set ;
    ( c1 c= {} iff c1 = {} ) by XBOOLE_1:3;
    hence ( c1 in bool {} iff c1 in {{} } ) by Def1, TARSKI:def 1;
  end;
  hence bool {} = {{} } by TARSKI:2;
end;
```

Its (probably redundant) natural-language explanation is that the powerset of the empty set is a singleton containing just the empty set. The symbol `in` used inside the proof denotes the set-theoretical membership, and the symbol `c=` used below denotes the set-theoretical inclusion. Note that the theorem `XBOOLE_1:3` [12]

```
theorem E3: :: XBOOLE_1:3
  for b1 being set holds ( b1 c= {} implies b1 = {} )
```

[12] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/xboole_1.html#T3

is just an implication, not equivalence. So the user might want to know, why the first inference [13]

```
( c1 c= {} iff c1 = {} ) by XBOOLE_1:3;
```

is logically valid. Clicking on its **by** keyword will reveal the following E-PROVER's proof [14] rendered by MML Query [15] (we rather recommend to check this (above given link) directly in a browser, since the linking cannot be seen here in text form). The very first axiom there is `reflexivity_r1_tarski`, stating the reflexivity of the inclusion predicate. This is a Mizar *property*, which the system uses automatically, assuming that it is obvious to the readers. With this axiom, the inference easily follows:

```
MML Query rendering of ATP proof steps

axiom: reflexivity_r1_tarski
A:step 1
for x1, x2 being set holds
x1 c= x1
------------------------------------------------------
conjecture: e1_10_1
A:step 7
   (c11001 c= {}
iff
   c11001 = {})
------------------------------------------------------
axiom: t3_xboole_1
A:step 8
for x1 being set
      st x1 c= {}
   holds x1 = {}
------------------------------------------------------
inference: assume_negation(7)
A:step 9
(c11001 c= {} & c11001 <> {} or c11001 = {} & not c11001 c= {})
------------------------------------------------------
inference: variable_rename(1)
A:step 11
for x3, x4 being set holds
x3 c= x3
------------------------------------------------------
inference: split_conjunct(11)
A:step 12
x1 c= x1
------------------------------------------------------
inference: fof_nnf(9)
A:step 22
(c11001 c= {} implies c11001 <> {}) &
 (c11001 c= {} or c11001 = {})
------------------------------------------------------
inference: split_conjunct(22)
A:step 23
(c11001 = {} or c11001 c= {})
------------------------------------------------------
inference: split_conjunct(22)
A:step 24
(c11001 = {} implies not c11001 c= {})
------------------------------------------------------
inference: fof_nnf(8)
A:step 25
for x1 being set
      st x1 c= {}
   holds x1 = {}
------------------------------------------------------
inference: variable_rename(25)
A:step 26
for x2 being set
      st x2 c= {}
   holds x2 = {}
------------------------------------------------------
inference: split_conjunct(26)
A:step 27
```

[13] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/zfmisc_1.html#E1:10_1

[14] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/_by/zfmisc_1/164_29.html

[15] Note that the constant $c1$ has been renamed by our system to $c11001$. This is actually $c1_{1001}$ in the HTML rendering, the subscript encodes the current Mizar proof level. This is a result of the MPTP system naming conventions, which need to provide unique name to every MPTP object.

```
(x1 <> {} implies not x1 c= {})
------------------------------------------------------
inference: csr(24,27)
A:step 29
not c11001 c= {}
------------------------------------------------------
inference: sr(23,29)
A:step 30
c11001 = {}
------------------------------------------------------
inference: rw(rw(29,30),12)
A:step 31
contradiction
```

2.2 Simple example further explained

Now we will explain the particular stages of creating and rendering of the ATP explanation.

2.2.1 Creation of the ATP problem

The problems are generated by the development version of the MPTP system. This is a system for translating Mizar into untyped first-order predicate logic, and encoding Mizar problems in a way suitable for solving by ATP systems. The recent (second) version of the system has been quite heavily tested (see [Urb06b]), and for problems which do not contain possible arithmetical evaluations (which will need further treatment) it now seems to provide all the information necessary for reproving Mizar inferences by ATPs. As mentioned above, MPTP encodes the Mizar types as predicates, and explicitly adds to problem specifications various kinds of information which is obvious to Mizar (like type hierarchy, or the reflexivity property mentioned above). The ATP problem specification (file named `zfmisc_1__164_29`, using the TPTP [SS98] format) is as follows:

```
% Mizar problem: e1_10_1,zfmisc_1,164,29
fof(reflexivity_r1_tarski, axiom,  (! [A, B] : r1_tarski(A, A)) ,
    file(tarski, r1_tarski), []).
fof(dt_k1_xboole_0, axiom, $true, file(xboole_0, k1_xboole_0), []).
fof(dt_c1_10_1, axiom, $true, file(zfmisc_1, c1_10_1), []).
fof(fc1_xboole_0, axiom, v1_xboole_0(k1_xboole_0),
    file(xboole_0, fc1_xboole_0), []).
fof(rc1_xboole_0, axiom,  (? [A] : v1_xboole_0(A)) ,
    file(xboole_0, rc1_xboole_0), []).
fof(rc2_xboole_0, axiom,  (? [A] :  ~ (v1_xboole_0(A)) ) ,
    file(xboole_0, rc2_xboole_0), []).
fof(e1_10_1, conjecture,
   (r1_tarski(c1_10_1, k1_xboole_0) <=> c1_10_1=k1_xboole_0) ,
   inference(mizar_bg_added, [],
   [reflexivity_r1_tarski, dt_k1_xboole_0, dt_c1_10_1, fc1_xboole_0,
   rc1_xboole_0, rc2_xboole_0, t3_xboole_1]), [file(zfmisc_1, e1_10_1)]).
fof(t3_xboole_1, axiom,
   (! [A] :  (r1_tarski(A, k1_xboole_0) => A=k1_xboole_0) ) ,
   file(xboole_1, t3_xboole_1), []).
```

This encoding is in more detail described in [Urb06b]. Note that there are more axioms than are actually needed for the proof above. This is because the MPTP algorithm for adding the "background knowledge" can be only approximative, and the main goal is to approximate it from the safe side, i.e. maintaining the completeness of the specification.

2.2.2 Creation of the ATP proof

The MPTP problems generated from the Mizar simple justifications are usually very easy for current ATP systems. We are using the latest version (0.91) of Stephan Schulz's E-PROVER, both because of its strength, and also because of its support of the TPTP standards, which allow us to use the TPTP tools for preprocessing

and postprocessing. Running the E-PROVER through SystemOnTPTP [16] yields the following TSTP proof object:

```
fof(1, axiom,![X1]:![X2]:r1_tarski(X1,X1),
    file('/tmp/SystemOnTPTP15237/zfmisc_1__e1_10_1.p', reflexivity_r1_tarski)).
fof(7, conjecture,(r1_tarski(c1_10_1,k1_xboole_0)<=>equal(c1_10_1, k1_xboole_0)),
    file('/tmp/SystemOnTPTP15237/zfmisc_1__e1_10_1.p', e1_10_1)).
fof(8, axiom,![X1]:(r1_tarski(X1,k1_xboole_0)=>equal(X1, k1_xboole_0)),
    file('/tmp/SystemOnTPTP15237/zfmisc_1__e1_10_1.p', t3_xboole_1)).
fof(9, negated_conjecture,
    ~((r1_tarski(c1_10_1,k1_xboole_0)<=>equal(c1_10_1, k1_xboole_0))),
    inference(assume_negation,[status(cth)],[7])).
fof(11, plain,![X3]:![X4]:r1_tarski(X3,X3),
    inference(variable_rename,[status(thm)],[1])).
cnf(12,plain,(r1_tarski(X1,X1)),inference(split_conjunct,[status(thm)],[11])).
fof(22, negated_conjecture,
    ((~(r1_tarski(c1_10_1,k1_xboole_0))|~(equal(c1_10_1, k1_xboole_0)))
     &(r1_tarski(c1_10_1,k1_xboole_0)|equal(c1_10_1, k1_xboole_0))),
    inference(fof_nnf,[status(thm)],[9])).
cnf(23,negated_conjecture,(c1_10_1=k1_xboole_0|r1_tarski(c1_10_1,k1_xboole_0)),
    inference(split_conjunct,[status(thm)],[22])).
cnf(24,negated_conjecture,(c1_10_1!=k1_xboole_0|~r1_tarski(c1_10_1,k1_xboole_0)),
    inference(split_conjunct,[status(thm)],[22])).
fof(25, plain,![X1]:(~(r1_tarski(X1,k1_xboole_0))|equal(X1, k1_xboole_0)),
    inference(fof_nnf,[status(thm)],[8])).
fof(26, plain,![X2]:(~(r1_tarski(X2,k1_xboole_0))|equal(X2, k1_xboole_0)),
    inference(variable_rename,[status(thm)],[25])).
cnf(27,plain,(X1=k1_xboole_0|~r1_tarski(X1,k1_xboole_0)),
    inference(split_conjunct,[status(thm)],[26])).
cnf(29,negated_conjecture,(~r1_tarski(c1_10_1,k1_xboole_0)),
    inference(csr,[status(thm)],[24,27])).
cnf(30,negated_conjecture,(c1_10_1=k1_xboole_0),
    inference(sr,[status(thm)],[23,29,theory(equality)])).
cnf(31,negated_conjecture,($false),
   inference(rw,[status(thm)],
     [inference(rw,[status(thm)], [29,30,theory(equality)]),
      12,theory(equality)])).
cnf(32,negated_conjecture,($false),inference(cn,[status(thm)],
                                             [31,theory(equality)])).
cnf(33,negated_conjecture,($false),32,['proof']).
```

Obviously, different ATP systems (or even the same ATP run with different settings) can produce different refutational proofs, and these proofs will generally also differ from the hypothetical "Mizar proof" (i.e., the steps done by the Mizar checker to justify the conjecture). Certainly, we might try to optimize the search in order to find e.g., the shortest (and thus hopefully the "best understandable") proofs. On the other hand, as noted above, these inferences are supposed to be quite easy for humans, and they turn out to be quite easy also for ATPs, so such (potentially quite resource-intensive) optimization is probably not worth its cost. A much better way to make the proofs more understandable is to spend some effort on their presentation, which is what we do in the following steps.

2.2.3 Preparing the ATP proof for MML Query rendering

The TSTP proof object is first transformed to the XML encoding of TSTP, by Petr Pudlák's and Geoff Sutcliffe's tool TPTP2XML. The XML listing would be too long to be included here, and it is available on our web site [17]. Then we apply our XSLT stylesheet `tstp2dli.xsl` [18] which translates the TPTP notation to the MML Query DLI (Decoded Library Item) format. MML Query is only used to process the DLI-encoded formulas, so another task of `tstp2dli.xsl` is to take care of the linking of formula names to the Mizar HTML pages, and linking of the ATP step

[16] http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTPFormMaker

[17] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html.930/_by_xml/zfmisc_1/164_29.xml

[18] http://kti.ms.mff.cuni.cz/cgi-bin/viewcvs.cgi/xsl4mizar/tstp2dli.xsltxt?view=markup

references. The result is again on our web site [19] .

2.2.4 Presentation with MML Query

As mentioned above, the MML Query DLI format is a notation for the Mizar semantic level. Therefore two tasks have to be done by MML Query:

- translate the formulas in MNF back into a human-friendly notation (i.e., usually compress it by using more logical connectives)
- translate Mizar constructors back into a human-friendly notation (i.e., find suitable Mizar symbols and their patterns, which correspond to the given constructor form)

First we explain the MNF transformation. A conjunction in MNF used in MML Query may have more than two conjuncts (the result of the associativity of conjunction in Mizar). Such situation is denoted here by $\&(\ldots)$. The reconstruction of richer set of connectives ($\exists$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$) tries to reverse the Mizar transformation algorithm and could be described by following rules:

$$\begin{gathered}\neg\forall_x\varphi \rightarrow \exists_x\neg\varphi \\ \neg\&(\varphi_1,\ldots,\varphi_{n-1},\neg\varphi_n) \rightarrow \&(\varphi_1,\ldots,\varphi_{n-1}) \Rightarrow \varphi_n \\ \neg\&(\neg\varphi_1,\ldots,\neg\varphi_n) \rightarrow \varphi_1 \vee \ldots \vee \varphi_n\end{gathered}$$

The reconstruction of $\Leftrightarrow$ is a bit more complicated. If formula has the form

$$\neg\&(\varphi_1,\ldots,\varphi_k,\varphi_{k-1},\ldots,\neg\varphi_n) \wedge \neg\&(\psi_1,\ldots,\psi_m)$$

and

$$\&(\neg\&(),\varphi_1,\ldots,\varphi_k) \equiv \&(\psi_1,\ldots,\psi_m)$$

then it is transformed to

$$\&(\varphi_1,\ldots,\varphi_k) \Leftrightarrow \neg\&(\varphi_{k+1},\ldots,\varphi_n)$$

The equivalence $\equiv$ is the smallest equivalence satisfying the conditions of double negation and single conjunction:

$$\neg\neg\varphi \equiv \varphi \qquad \text{and} \qquad \&(\varphi) \equiv \varphi$$

The compression of quantifiers is also applied and formulas of the form $\forall_x(\varphi \Rightarrow \psi)$ are presented as `for x st` φ `holds` ψ. The indenting and breaking of long formulas is applied for better readability. The above rules assume that transformed formulas do not include double negations nor nested or single conjunctions (this is the case of formulas generated from Mizar). So, the transformation of an arbitrary formula expressed with $\wedge$, $\neg$, $\top$, and $\forall$ requires additional pruning at beginning. This has been implemented to handle our ATP-generated data.

The translation of Mizar constructors into the human-friendly notation uses the "rule of first available notation". Generally, multiple synonyms (or antonyms) can exist as user symbols corresponding to one Mizar constructor (in other words,

[19] http://lipa.ms.mff.cuni.cz/~urban/xmlmml/html_bytst/_by_dli/zfmisc_1/zfmisc_1__164_29.dli

when a Mizar author introduces a synonym, it exists only on the presentation level). Since in the general case (like the one when data come from ATPs) we have no other information than the constructor format, it is reasonable to present the constructor using just the first human symbol found in the MML, which corresponds to that constructor. More special rules can be (and are) used by MML Query, when additional information is available, e.g., about the Mizar articles from which the constructor encoding comes.

The result of the final MML Query rendering step on our example is already shown above, it is the final explanation of the "simple justification" inference that is presented to the reader.

3 MML Query presentation versus the XML-based HTML presentation of full articles

The above described by-explanation system, which we have implemented for the 35 initial Mizar articles, uses two different techniques for presentation of the Mizar semantic level. The presentation of the by-explanations is done by the MML Query "artificial intelligence" reconstruction of a possible user notation (described in 2.2.4). With no other information added to the constructor encoding, there is really no other choice. On the other hand, for the XML-based HTML presentation of full articles, to which the MML Query explanations are linked, a lot of further information is available, namely from the Mizar parser, which was used to process the original Mizar text. Even though the purpose of the Mizar XML format is to primarily contain the semantic information, the XML format allows for easy addition of the original presentation information. This feature has been added to Mizar some time ago, and the XML produced by recently distributed Mizar versions already contains this additional presentational information.

It is therefore quite likely, that in many cases the presentations of the same Mizar formula by these two systems will differ. We don't think that in the particular case of the system presented above this is necessarily harmful. While the goal of the XML-based HTML presentation is to achieve high resemblance to the original article, with as much additional semantic information as possible, the goal of the MML Query presentation is to present pieces of Mizar text in a uniform and predictable way. In this sense the MML Query presentation can be thought of as a tool for strong uniform formatting of Mizar. While this difference may be initially surprising for a reader, when he first uses the by-explanation functionality, this might also lead to his deeper understanding of the presentation process, and of Mizar itself. Should this become a serious problem, we can always offer to the user to apply the MML Query formatting to the whole XML-based article presentation.

4 Conclusions and Future Work

We have provided an initial implementation of a system explaining the Mizar atomic "simple justification" inferences, and demonstrated it on 35 initial Mizar articles. For this, we have used or newly developed a chain of tools working on the Mizar semantic level. The systems could be already now extended to a much larger por-

tion of MML, since the ATP success rate on solving nonnumerical Mizar "simple justification" problems is very high (above 90% with 4s timelimit and one ATP used in a push-button manner, and above 99% with more sophisticated, but still fully automated methods described in Section 4.3. of [Urb06b]). However we still do not have a satisfactory algorithm for serious transformation of the ATP solutions on heavily typed Mizar articles to the Mizar typed semantic level. It is possible already now to render such solutions with MML Query as mentioned above, and it probably would be useful purely for the explanation purpose. However such rendering will be incorrect from the Mizar parser's point of view, which requires that variables are qualified with proper types when typed functors or predicates are applied to them. A transformation of the ATP untyped solutions to Mizar-acceptable typed solutions is probably feasible, and it is an interesting line of further research. Obviously all the tools participating in our chain can be improved too. One interesting idea is to have all of the systems participating in the chain working in real time, passing the solutions to each other. Such functionality will probably be developed quite soon, and generally used for providing ATP support to Mizar authors. From this point of view, providing the current by-explanation functionality can be thought of as a test-bed, making the way for more ambitious ATP-for-Mizar applications.

5 Acknowledgments

Thanks to Stephan Schulz, Geoff Sutcliffe, and Petr Pudlák for developing and providing the ATP systems and metasystems participating in our tool chain. Geoff Sutcliffe has also first suggested the "click for ATP" functionality in the Mizar HTML presentation. This work was supported by a Marie Curie International Fellowship within the 6^{th} European Community Framework Programme.

References

[ABT04] Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors. *Mathematical Knowledge Management, Third International Conference, MKM 2004, Bialowieza, Poland, September 19-21, 2004, Proceedings*, volume 3119 of *Lecture Notes in Computer Science*. Springer, 2004.

[BR03] Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in MML. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 119–132. Springer, 2003.

[BU04] Grzegorz Bancerek and Josef Urban. Integrated semantic browsing of the Mizar Mathematical Library for authoring Mizar articles. In Asperti et al. [ABT04], pages 44–57.

[Byl89] Czeslaw Bylinski. Some basic properties of sets. *Formalized Mathematics*, 1(1), 1989.

[Jas34] S. Jaskowski. On the rules of suppositions. *Studia Logica*, 1, 1934.

[McC94] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, Illinois, 1994.

[NB04] Adam Naumowicz and Czeslaw Bylinski. Improving mizar texts with properties and requirements. In Asperti et al. [ABT04], pages 290–301.

[Pel99] F. J. Pelletier. A brief history of natural deduction. *History and Philosophy of Logic*, 20:1 – 31, 1999.

[RT99] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. *J. Autom. Reasoning*, 23(3-4):197–234, 1999.

[Rud92] P. Rudnicki. An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs*, pages 311–332. Chalmers University of Technology, Bastad, 1992.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *Journal of AI Communications*, 15(2-3):91–110, 2002.

[Sch02] S. Schulz. E – a brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[Urb04] Josef Urban. MPTP - motivation, implementation, first experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.

[Urb05] Josef Urban. MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. *Journal of Applied Logic*, 2005. Article In Press, doi:10.1016/j.jal.2005.10.004, available online at http://ktiml.mff.cuni.cz/∼urban/mizmode.ps.

[Urb06a] Josef Urban. MoMM - fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.

[Urb06b] Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *Journal of Automated Reasoning*, 2006. Accepted for Publication.

[Urb06c] Josef Urban. XML-izing Mizar: making semantic processing and presentation of MML easy. In Michael Kohlhase, editor, *MKM 2005*, volume 3863 of *Lecture Notes in Artificial Intelligence*, pages 346–360. Springer, 2006.

[WBH+02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic. SPASS version 2.0. In *CADE*, pages 275–279, 2002.

[Wei01] C. Weidenbach. *Handbook of Automated Reasoning*, volume II, chapter SPASS: Combining Superposition, Sorts and Splitting, pages 1965–2013. Elsevier and MIT Press, 2001.

[Wie00] Freek Wiedijk. CHECKER - notes on the basic inference step in Mizar. available at http://www.cs.kun.nl/∼freek/mizar/by.dvi, 2000.

An Interactive Derivation Viewer

Steven Trac, Yury Puzis, Geoff Sutcliffe[1]

Department of Computer Science
University of Miami
Coral Gables, USA

Abstract

This work describes the *Interactive Derivation Viewer* (IDV) tool for graphical rendering of derivations that are written in the TPTP language. IDV provides an interactive interface that allows the user to quickly view various features of the derivation. A particularly novel feature of IDV is its ability to provide a synopsis of a derivation by identifying *interesting* lemmas within a derivation, and hiding less interesting intermediate formulae. IDV is deployed online as part of the SystemOnTPTP interface, thus providing ready access via any web browser.

Keywords: Derivation viewer, Proof synopsis

1 Introduction

The proofs output by automated reasoning systems provide useful information to system users, e.g., the proof structure, lemmas that may be useful in future proofs, which axioms are most used, etc. Even derivations that do not form completed proofs are of interest, as they may provide insights leading to changes in the problem formulation or the system application, that result in a proof being found - automated reasoning systems are often debugged in this way. However, the proofs output by automated reasoning systems are often unsuitable for human consumption. For first-order automated theorem proving (ATP) systems, the reasons include:

- The conversion of problems stated in "natural" first-order form (FOF) to clause normal form (CNF).
- The use of proof by contradiction, which introduces formulae that are not logical consequences of the axioms.
- The use of fine grained inference steps, such as binary resolution, that exaggerate the size of a proof.

Several types of tools have been developed to make the output from ATP systems easier for humans to understand. These include graphical renderings of derivations

[1] Email: strac@mail.cs.miami.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

[10], structuring of proofs by identification of lemmas [1], translation of resolution refutation proofs to natural deduction proofs [4], and full translation of proofs to natural language form [2]. This work describes the *Interactive Derivation Viewer* (IDV) tool for graphical rendering of derivations that are written in the TPTP [13] language [12]. IDV provides an interactive interface that allows the user to quickly view various features of the derivation. A particularly novel feature of IDV is its ability to provide a synopsis of a derivation by identifying *interesting* lemmas within a derivation, and hiding less interesting intermediate formulae. IDV is deployed online as part of the SystemOnTPTP interface [11], thus providing ready access via any web browser.

Section 2 describes the basic IDV tool and it's rendering process. Section 3 describes the production of proof synopses. Section 4 explains how IDV is deployed on the web, and provides an illustrative application. Section 5 concludes and discusses future developments planned for IDV.

2 Basic IDV

A derivation is a directed acyclic graph (DAG) whose leaf nodes are formulae (possibly derived) from the input problem, whose interior nodes are formulae inferred from parent formulae, and whose root nodes are the final derived formulae. For example, a CNF refutation proof is a derivation whose leaf nodes are the clauses formed from the axioms and the negated conjecture, and whose root node is the $false$ clause. The information required to record a derivation is, minimally, the leaf formulae, and each inferred formula with references to its parent formulae. More detailed information that may be recorded and useful includes: the role of each formula, e.g., axiom, conjecture, plain derived, etc; the name of the inference rule used in each inference step; sufficient details about each inference step to deterministically reproduce the step; and the semantic relationship of each inferred formula with respect to its parents, e.g., logical consequence, counter theorem, etc. The TPTP language is sufficient for recording all this, and more.

A derivation written in the TPTP language is a list of annotated formulae. Each annotated formula contains a name, a role, the logical formula, a source record, and a field for recording arbitrary useful information, as required for user applications. The source of each inferred formula is an `inference` record containing the inference rule name, a `status` record containing the semantic relationship of the formula to its parents as an SZS ontology value [14], and a list of references to its parent formulae.

IDV takes a derivation in the TPTP language and renders the DAG using Java's `Swing` components. IDV can run as a standalone application, or as a web browser applet; this description focuses on the web option, because it provides ready (remote) access to IDV without any installation required.

2.1 Interface

Figure 1 shows the rendering of the derivation output by the ATP system EP 0.91 [9] for the TPTP problem `PUZ001+1`.[2] The IDV window is divided into three panes: the top control strip pane provides control buttons and sliders, the main middle pane shows the rendered DAG, and the bottom pane gives the text of the annotated formula for the node pointed to by the mouse.

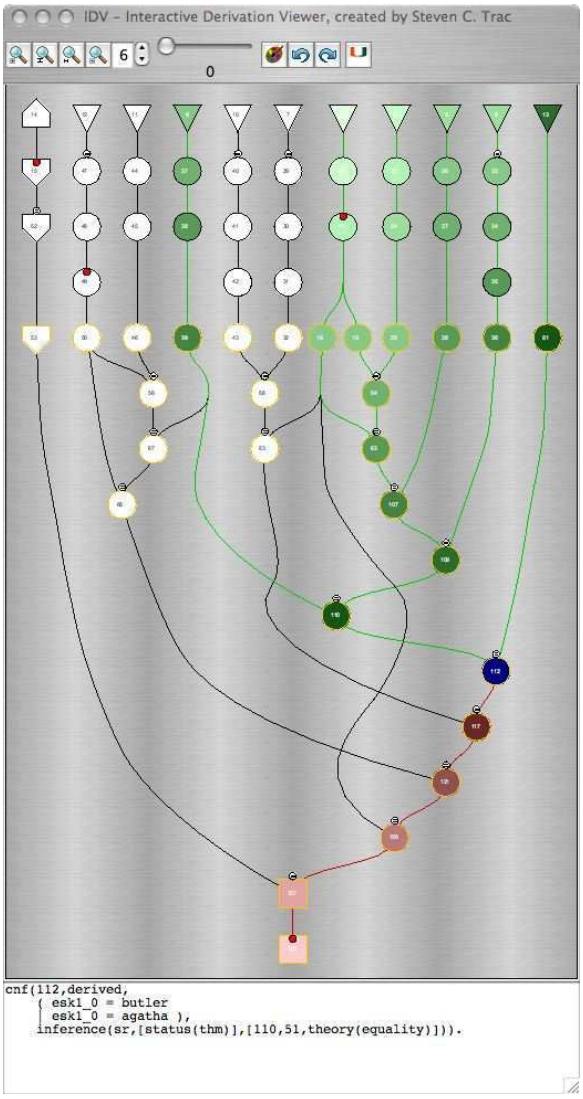


Fig. 1. EP's Proof by Refutation of `PUZ001+1`

The buttons and sliders in the control strip pane are, from left to right:

- Zoom in - zooms in 50%
- Fit vertical - scales the rendering to fit the height of the middle pane
- Fit horizontal - scales the rendering to fit the width of the middle pane

[2] `PUZ001+1` is the "Aunt Agatha" problem, which describes a scenario in which three people live in a mansion, and Aunt Agatha is killed. The goal is to prove that Aunt Agatha killed herself. All TPTP problems, their solutions, and IDV renderings of the solutions, are available online via `http://www.tptp.org/` - follow the Problems link to reach the problems, the TSTP link to reach the solutions, and the View IDV Tree link at the top of any solution page (that has the solution in TPTP format) to generate the IDV rendering.

- Zoom out - zooms out 50%
- Display height - sets the number of text lines in the bottom pane
- Synopsis level - sets the minimal interestingness level for display - see Section 3.
- Redraw - redraws the derivation. This is typically used after extracting a synopsis - see Section 3.
- Synopsis undo - sets the minimal interestingness level to its previous value.
- Synopsis redo - sets the minimal interestingness level to its next value, after any undo steps.
- About button

The rendering of the derivation DAG uses shape and color to visually provide information about the derivation. Each node corresponds to a formula in the derivation, with FOF nodes outlined in black and CNF nodes outlined in orange. The role of the formulae is indicated by the shape of the node: triangle for axioms, hexagons for lemmas, inverted trapezium for hypotheses, house for conjectures, inverted house for negated conjectures (as done when converting a FOF problem to CNF), circle for plain derived formulae, and square for $false$ formulae. A node may be annotated above with a = sign in a circle to indicate that equality reasoning was used in its inference, e.g., a paramodulation inference. A node may be annotated inside at the top with a red circle to indicate that the formula is not a logical consequence of its parents, e.g., in Skolemization and splitting inferences, as indicated by the SZS status. A node may be annotated below with a red triangle to indicate that it is the parent of a splitting inference, e.g., an explicit split as implemented by SPASS [15] or a pseudo-split as implemented by Vampire [7,8] or E [9].

The user can interact with the derivation rendering in two ways. First, moving the mouse over any node causes the annotated formula corresponding to the node to be shown in the bottom pane. At the same time, the moused-over node is highlighted in blue, all nodes leading down from leaf nodes into the moused-over node are highlighted in green, and all nodes leading down from the moused-over node to root nodes are highlighted in red. The effect is evident in Figure 1. The green highlighting shows from which formulae the moused-over node is derived, and the red highlighting shows which formulae are derived from the moused-over node. The intensity of the highlighting decreases according to the minimal path length from the moused-over node to the highlighted node. This allows easy differentiation between closer and more distant ancestors and descendants. A particularly useful effect is to identify which axioms (leaf nodes) are the closest ancestors. The second form of interaction is to click on any node. This creates a pop-up window containing the annotated formulae of the clicked node and its parents, as shown in Figure 2. The annotated formula of the clicked node is shown twice once above the parents and again below, allowing for bidirection reading of the inference step.

2.2 Implementation

IDV reads in a derivation in TPTP format. It is sensitive to the form of the formulae, either FOF or CNF. The rendering is performed in two phases: the first phase determines the layout of the DAG nodes and edges, and the second phase implements the graphical display of the derivation DAG.

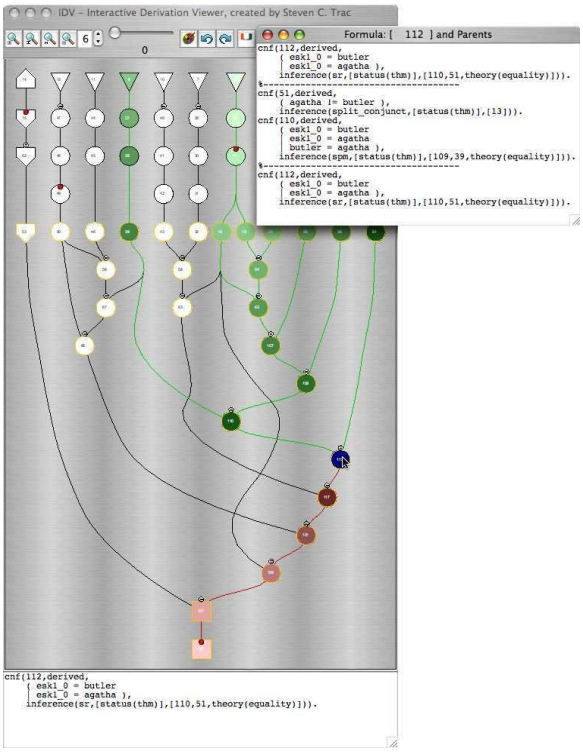


Fig. 2. Pop-up in EP's Proof by Refutation of PUZ001+1

The layout of the DAG is determined in a five-pass algorithm, similar to that in [3]. The first pass assigns a rank to each node. The rank is the level of the node in the rendered tree, with the top row containing the nodes at level 1, and increasing downwards. The rank is used in the third pass to determine the Y coordinate of each node. The first row of FOF nodes (leaf nodes) are placed in the first rank. A depth first search (DFS) is then used to assign increasing rank to the rest of the FOF nodes. Next the first row of CNF nodes are placed in the rank above the maximum FOF node rank. Finally, the DFS algorithm is run again to assign increasing rank to the rest of the CNF nodes. After ranks are assigned to all nodes, the edges are partitioned as follows: If an edge connects two nodes that are more than one rank apart, the edge is replaced by a chain of virtual nodes and edges. The virtual nodes are given incremental ranks between the two end nodes' ranks. If a non-virtual node has more than one chain of virtual nodes leading down from it, the chains are combined as far as possible, dividing immediately above the end nodes of the chains.

The second pass directly follows the algorithm from [3], setting the left-to-right vertex order within ranks by an iterative heuristic incorporating a weight function and local transpositions to reduce edge crossings. The introduction of the virtual nodes at each rank guarantees that edge crossings can only occur between adjacent ranks.

The third pass sets an initial X coordinate and final Y coordinate for each node. The rank with the largest number of nodes determines the maximum width of the graph. With the left-to-right vertex ordering within ranks from the second pass, equidistant X coordinates are given to the nodes in each rank, between 0 and

maximum width of the graph. The final Y coordinate is based linearly on the nodes' ranks.

The fourth pass finds the optimal X coordinate for each node. For this pass spring embedding is used.[3] Spring embedding is a graph drawing technique that models a graph as a system of springs and then uses energy minimization to space the nodes. The following forces are balanced: edge spring force for keeping edges at a certain length, node-to-node repulsive forces to keep nodes from being too close, gravity force that keeps all edges pointing downwards, and repulsive boundary forces to keep the nodes from spreading too far apart horizontally. After the fourth pass the X and Y coordinates are fixed - the nodes cannot be moved by user interaction.

The fifth pass generates Bezier curves to draw edges between nodes. If two non-virtual nodes are connected by a chain of virtual nodes, then the chain of virtual nodes is used to plot the points of the Bezier curve.

The layout determined by the first phase does not guarantee that nodes will not overlap (or hence, given the use of virtual nodes to guide edge generation, that edges will not pass through nodes). The extent to which node overlaps are avoided is determined by the number of iterations in the spring embedding. The number of iterations in the current implementation has been found to be sufficient to avoid most overlaps. After the layout has been determined, the interface and DAG are rendered.

IDV is implemented in Java, mainly using basic `Swing` components. The TPTP formulae are read in using `StreamTokenizer`. The IDV window is a `JFrame`, and the rendering is a `JPanel`. A `MouseMotionListener` is used in the `JPanel` to detect when the mouse moves over a node, to implement the node coloring feature. A `MouseListener` is used in the `JPanel` to detect when a node is clicked, to implement the pop-up window feature. The `JFrame` is implemented with `ActionListener` and `ChangeListener` to detect the user's manipulations in the control strip.

3 Derivation Synopses

As mentioned in Section 1, one of the features of derivations output by ATP systems is the use of fine grained inference steps such as binary resolution, which exaggerate the size of a proof. Derivations output by humans typically use coarser grained inference steps, leaving intermediate steps "to the reader". The inferred formulae of such coarser grained inference steps are logical consequences of their leaf ancestors, at various levels of saliency - humans often single out certain of the logical consequences to be specifically designated as lemmas. By considering only those logical consequences above a certain level of saliency (hiding those below that level), a synopsis of the detailed derivation is formed. In a synopsis the lowest visible ancestors of a hidden formula become the parents of the highest visible descendents. A synopsis hides the fined grained inference steps and the intermediate formulae, thus making is easier for a user to grasp an overview of the proof. The user may later choose to examine the details. Synposes may similarly be used to summarize extremely large derivations.

[3] Thanks to Christian Duncan for providing the original spring embedding code.

IDV is able to form a synopsis of a proof by CNF refutation. This is achieved by rating the *interestingness* of inferred CNF formula, and hiding the nodes whose formula rating is below a user specified threshold. The interestingness rating of inferred CNF formulae is computed by the AGInT system [6] - see Section 3.3, and the user sets a threshold using the slider in the control strip pane.

3.1 Interface

The interestingness of each formula is a value in the range 0.0 to 1.0. Some formulae have a preset interestingness rating: leaf formulae are set at 1.0, the topmost CNF formula are set at 1.0, the intermediate formulae between leaf FOF formulae and topmost CNF formulae are set at 0.0, all formulae derived from the negation of a conjecture are set at 1.0, and root formulae are set at 1.0. The interestingness of values for the other formulae, i.e., the internal CNF formulae of the derivation, are computed by the AGInT system, as described in Section 3.3. Initially the threshold slider in the top pane is set to an interestingness value of 0.0, and all nodes are displayed in the rendering. As the slider is moved up the interestingness threshold increases, and nodes whose formula rating is below the threshold are hidden. Figure 3 shows the derivation in Figure 1, with a interestingness threshold of 0.5.

After extracting a synopsis it is possible to zoom in, rendering only the visible nodes. This is done in IDV with the redraw button in the control strip. Figure 4 shows the synopsis derivation rendering of Figure 3. After a redraw the threshold slider may be moved and the derivation redrawn again, to produce a different level of synopsis. Note that after a redraw, if the threshold is moved to below the interestingness level used for the redraw, the hidden nodes do not immediately become visible - another redraw is required. The user is warned of this state by the threshold value being shown in red. Sequences of redraws can be undone and redone using the synopsis undo and redo buttons.

While using the slider to adjust the interestingness level, the layout of the nodes does not change - simply more or less of the nodes are hidden. This provides a identity mental map of the derivation (a *mental map* is the user's memory of the rendering [5]). When redrawing a synopsis it important to maintain the mental map as far as possible. To this end, all nodes that are not hidden in a synopsis are kept in the same order as in the original. The Bezier curves that connect the visible nodes are recomputed, but maintain the same form as in the original.

As mentioned in Section 2.1, when a node is clicked a pop-up window appears containing the annotated formula of the node and its parents. After a reddraw, the parents shown in a pop-up window are the parents of the formula in this rendering, i.e., they might not be the formulae's original parents. If some parent information is different than the original, then the pop-up window informs the user of this.

3.2 Implementation

Interestingness ratings are stored in a record in the `useful_info` fields of annotated formulae. There are two ways for formulae to have interestingness ratings. First, the annotated formulae input to IDV may already have interestingness values. Second, the input formulae do not have interestingness ratings, and the AGInT system has to

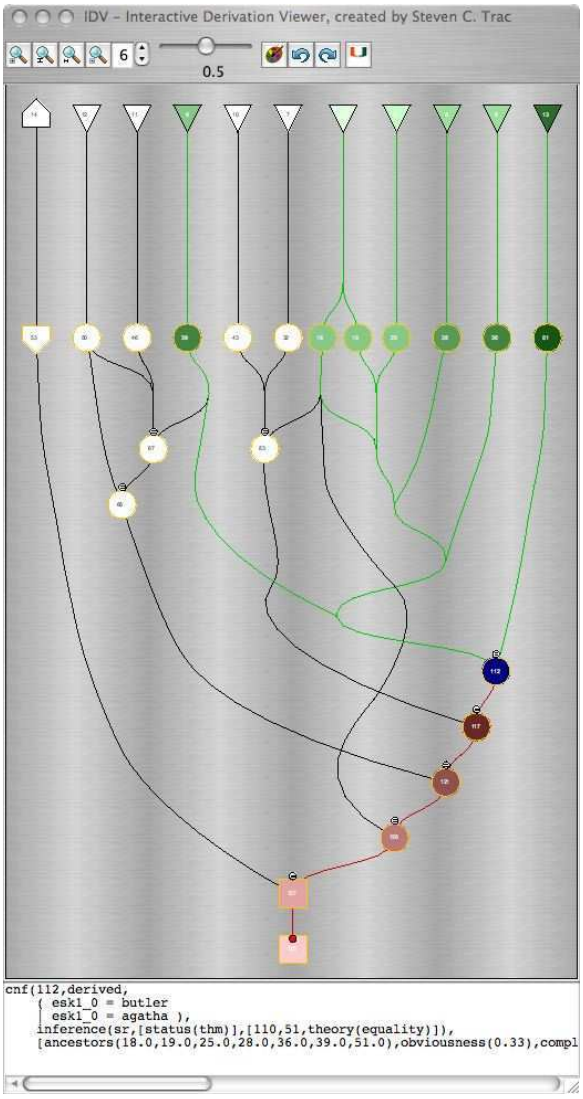


Fig. 3. Interesting nodes of EP's Proof by Refutation of PUZ001+1

be called by IDV. In this case AGInT is called as soon as the user uses the threshold slider in the control pane.

When the redraw button is clicked by the user, the derivation synopses is rendered as follows: First, non-virtual nodes in the DAG are set to be interesting if their interestingness rating is greater than the threshold value, and all virtual nodes are set to be uninteresting. Each rank is then checked to see if it contains any interesting nodes. If a rank contains at least one interesting node the rank is retained, otherwise the rank is empty and all nodes in ranks below are moved up a rank (i.e., their rank is decremented). The original rank of each node is stored for redrawing purposes. After the ranks are updated the Y coordinates of the nodes in the retained ranks are updated, as in Section 2.2. Finally, the Bezier curves are updated to uniquely connect each interesting node to its closest interesting ancestors, which are found using a DFS search up the DAG. All uninteresting nodes remain hidden after a redraw.

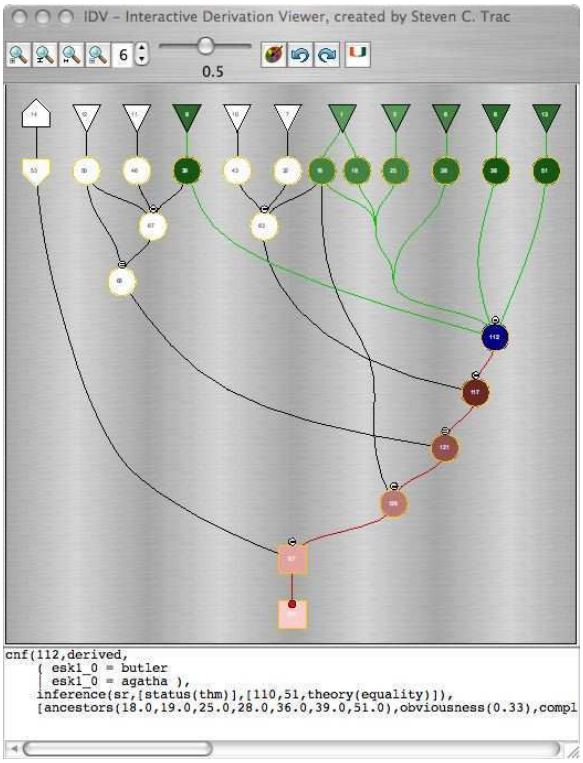


Fig. 4. Synopsis of EP's Proof by Refutation of PUZ001+1

When the synopsis undo/redo button is clicked, the current interesting threshold value changes to the last value pushed onto the undo/redo stack and above redraw procedure is called.

3.3 Interestingness Ratings

The interestingness ratings of derived CNF formulae in a derivation are computed by the *runtime filter* and *static ranker* components of the AGInT system. AGInT is a system that discovers interesting theorems of a given set of axioms. AGInT uses a deductive approach to discovery - it uses an ATP system to generate CNF logical consequences of the given set of axioms, filters the logical consequences to extract interesting theorems, and then computes an interestingness rating for each theorem. This basic process takes place in the context of an outer level control loop that regularly refocuses the generation of logical consequences, thus enabling AGInT to proceed deeply into the search space of logical consequences. Details are given in [6].

In the context of IDV, the derived CNF formulae of a derivation are given to AGInT as the logical consequences of the topmost CNF formulae (i.e., the topmost CNF formulae are considered to be the axioms from which the formulae are derived). AGInT's runtime filter and static ranker are used to compute interestingness values for the formulae. Figure 5 shows the combined architecture of these two components.

The task of the runtime filter is to aggressively filter out and discard boring formulae. Each formula must first pass the pre-processor, and must then pass the majority (i.e., at least four) of the seven filters: obviousness, weight, complexity,

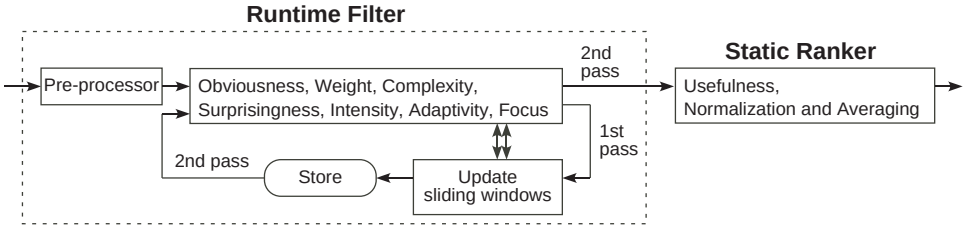


Fig. 5. Architecture of AGInT's Runtime Filter and Static Ranker

surprisingness, intensity, adaptivity, and focus. Each filter maintains a sliding window defined by the best distinct scores from the filter's evaluation of the formulae seen so far. The upper and lower bounds of each window are initialized to the worst possible score for that filter. If an incoming formula is scored equal to or better than the lower bound, it passes the filter, and the score is used to update the window. Initializing the upper and lower bounds to the worst possible score allows all formulae through until the window starts sliding up. As a result some boring formulae early in the stream may pass the runtime filter. Therefore the formulae that pass the runtime filter in the first pass are stored, and after all formulae have been processed the stored formulae are filtered again, with the windows fixed from the first pass. This removes any that do not meet the final lower bounds.

The individual filters are as follows:

Pre-processor: The preprocessor detects and discards obvious tautologies, e.g., clauses that contain an atom and it's negation, and clauses containing a *true* atom.

Obviousness: Obviousness estimates the difficulty of proving a formula. The obviousness score of a formula is the number of inferences in its derivation. A higher score is better.

Weight: Weight estimates the effort required to read a formula. Formulae that contain very many symbols (variables, function and predicate symbols) are less interesting. The weight score of a formula is the number of symbols it contains. A lower score is better.

Complexity: Complexity estimates the effort required to understand a formula. Formulae that contain very many different function and predicate symbols, representing many different concepts and properties, are less interesting. The complexity score of a formula is the number of distinct function and predicate symbols it contains. A lower score is better.

Surprisingness: Surprisingness measures new relationships between concepts and properties. Formulae that contain function and predicate symbols that are seldom seen together in a formula are more interesting. The symbol-pair surprisingness score of a pair of symbols is the number of axioms that contain both symbols divided by the number of axioms that contain either symbol. The surprisingness score of a formula is the sum of the symbol-pair surprisingness scores, over all pairs of distinct symbols in the formula. A lower score is better.

Intensity: Intensity measures how much a formula summarizes information from the leaf ancestors in its derivation tree. The plurality score of a formula (or set of formulae) is number of function and predicate symbols in the formula divided by the number of distinct function and predicate symbols in the formula. The intensity score of a formula is the plurality of its leaf ancestors divided by the plurality of

the formula itself. A higher score is better.

Adaptivity: Adaptivity measures how tightly the universally quantified variables of a formula are constrained. The adaptivity score of a clause is the number of distinct variables in the clause divided by the number of variable occurrences in the clause. A lower score is better.

Focus: Focus measures the extent to which a formula is making a positive or negative statement. Let FPL and FNL be the fractions of positive and negative literals in a clause. The focus score of a clause is $1 + FPL * log_2(FPL) + FNL * log_2(FNL)$. Clauses with up to three literals are considered to have perfect focus because their polarity distribution is limited. A higher score is better.

The formulae that pass the runtime filter are considered to be interesting. The task of the static ranker is to compute a final interestingness rating for the formulae. This is done in two phases: first a usefulness score is computed for each formula, and second, all the scores are individually normalized and then averaged.

Usefulness: Usefulness measures how much an interesting formula has contributed to proofs of further interesting formulae, i.e, its usefulness as a lemma. The usefulness score of a formula is the ratio of its number of interesting descendents (i.e., descendents that have passed the runtime filter) over its total number of descendents. A higher score is better.

Normalization and Averaging: The scores of the formulae, from each of the runtime filter and static evaluations, are normalized into the range 0.0 to 1.0. The formulae with the worst score are given a final score of 0.0, the formulae with the best score are given a final score of 1.0, and all other scores are linearly interpolated in between. If the worst and best score of a particular filter are equal, then that filter does not differentiate between the formulae, and those scores are removed. The remaining scores of each formula are averaged to produce a final interestingness rating.

4 Deployment and an Application

IDV is deployed online as part of the SystemOnTPTP interface at

`http://www.tptp.org/cgi-bin/SystemOnTPTPFormMaker`

The IDV code is wrapped as a web browser applet, and all computation for the rendering is done on the client side. The annotated formulae that constitute the derivation to be rendered may be passed to the applet as a parameter within the `<APPLET>` tags in the encompassing web page, or retrieved from a URL specified as a parameter within the `<APPLET>` tags. The AGInT code is deployed as a server side `cgi-bin` script, and is invoked by the IDV code via a `POST` call when interestingness ratings are required. Figure 6 shows the deployment architecture.

IDV has been used to analyze proofs of theorems, to identify key steps in the proofs. As an example EP's proof of the TPTP problem `SET615+3` is considered. This problem proves that for any three sets X, Y, and Z, $(X \cup Y) \backslash Z = (X \backslash Z) \cup (Y \backslash Z)$. Figure 7 shows EP's derivation DAG - clearly a hairy beast which is hard to comprehend as a whole. Figure 8 shows a synopsis of the derivation. It is very easy to see which nodes are key points in the synopsis, e.g., the one with the blue (darkest) coloring has the formula $X = (X \cup Y) \backslash (Y \backslash X)$. Another key node has the

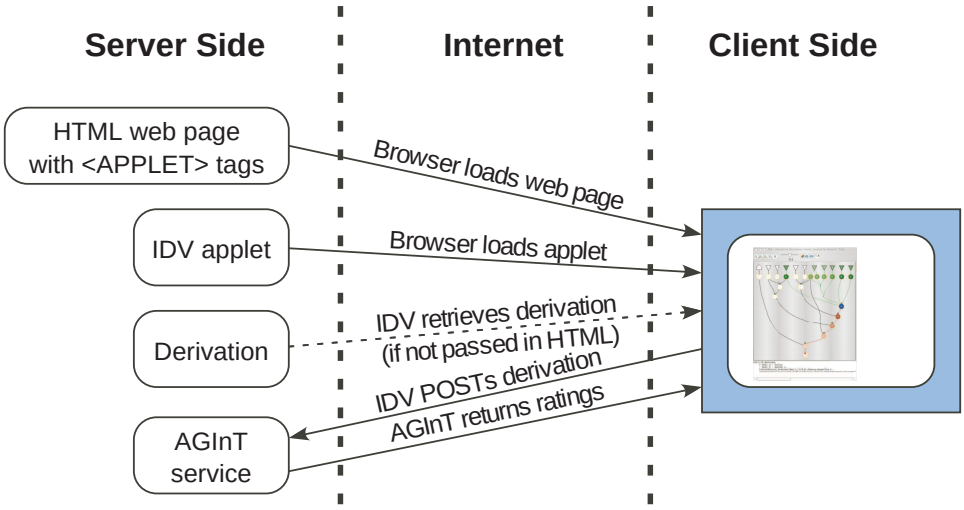


Fig. 6. Deployment of IDV

formula $X \cup (Y \backslash Z) = (X \cup Y) \backslash (Z \backslash X)$.

5 Conclusion

This paper has presented the design, implementation, deployment, and application of an interactive derivation viewer, implemented as the IDV tool. IDV provides strong visual information showing the structure of a derivation, with original details available as text. IDV provides interactive features that enable a user to visually highlight and examine salient parts of a derivation. In particular, the ability to extract proof synopses sets IDV apart from other existing derivation viewers. The use of "interestingness ratings", which are artificially intelligently determined, to provide a sliding scale of proof synopsis, is particularly powerful and certainly highly novel. The online deployment makes IDV easily available to users (who use the TPTP language for their derivations), without any need for software installation.

Future work planned for IDV includes finer grained synopsis of the FOF to CNF parts of derivations, which are currently considered to be not interesting at all. Future work on the implementation includes tighter integration with the SystemOnTPTP interface, so that interestingness ratings are computed in advance of their need, improving the performance on extremely large derivations, and improving the Bezier curve drawing in synopses of very large derivations. When the features have been optimized and implementation is stable, user evaluation will also be desirable.

References

[1] J. Denzinger and S. Schulz. Recording, Analyzing and Presenting Distributed Deduction Processes. In H. Hong, editor, *Proceedings of the 1st International Symposium on Parallel Symbolic Computation*, number 5 in Lecture Notes Series on Computing, pages 114–123. World Scientific Publishing, 1994.

[2] A. Fiedler. P.rex: An Interactive Proof Explainer. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 416–420. Springer-Verlag, 2001.

[3] E. Gansner, E. Koutsofios, S. North, and K-P. Vo. A Technique for Drawing Directed Graphs. *Software Engineering*, 19(3):214–230, 1993.

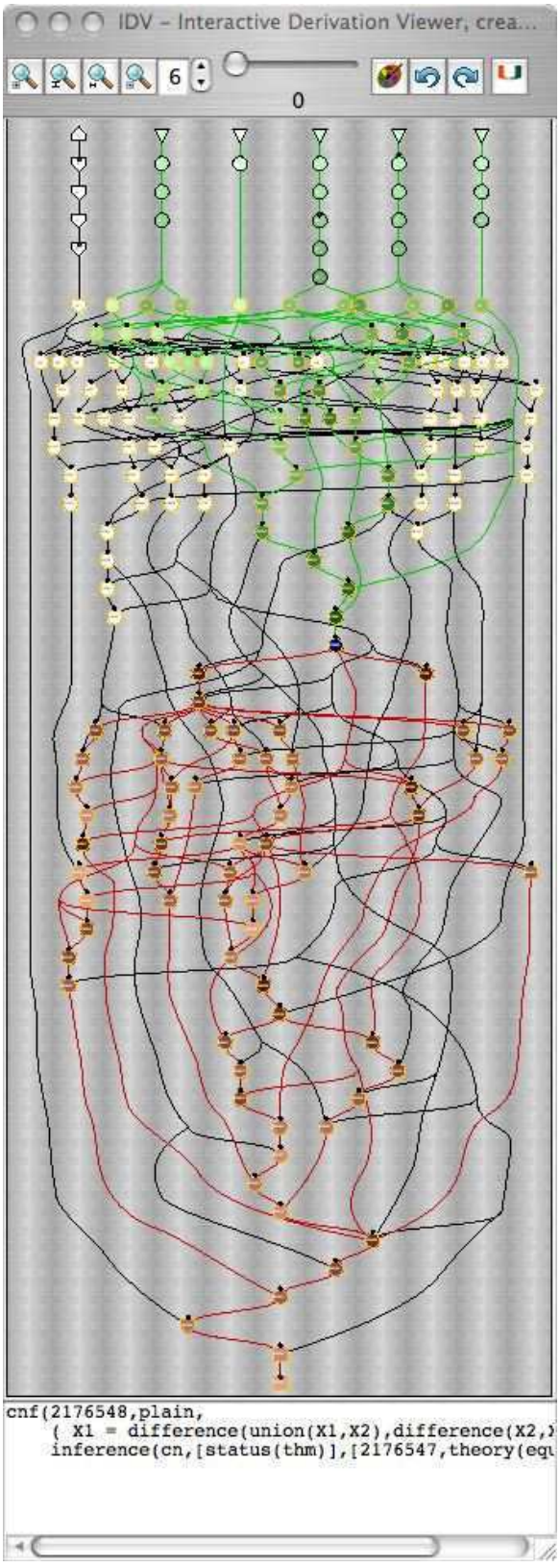


Fig. 7. EP's Proof by Refutation of SET615+3

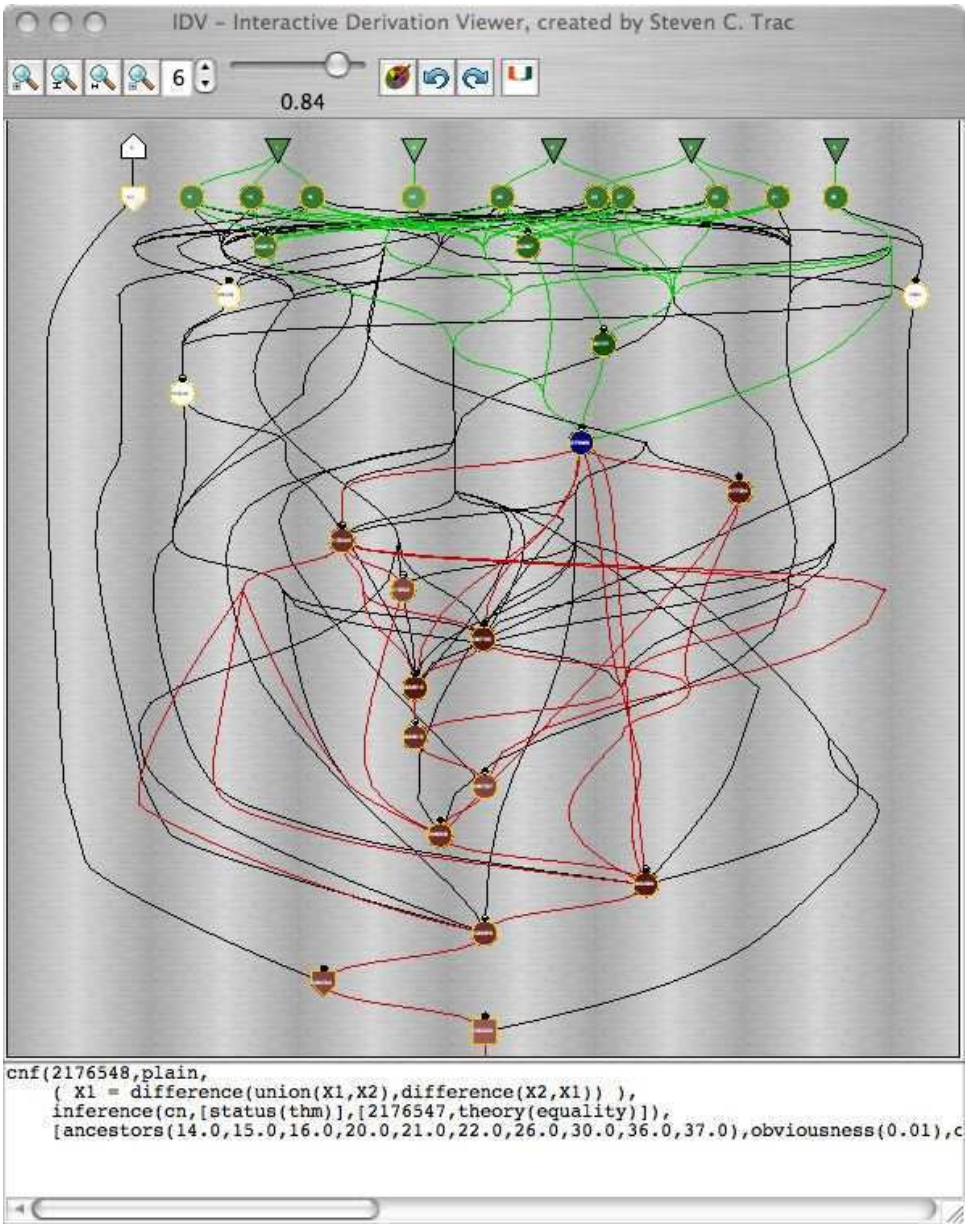


Fig. 8. Synopsis of EP's Proof by Refutation of `SET615+3`

[4] A. Meier. System Description: TRAMP - Transformation of Machine-Found Proofs into Natural Deduction Proofs at the Assertion Level. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 460–464. Springer-Verlag, 2000.

[5] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[6] Y. Puzis, Y. Gao, and G. Sutcliffe. Automated Generation of Interesting Theorems. In G. Sutcliffe and R. Goebel, editors, *Proceedings of the 19th International FLAIRS Conference*. AAAI Press, 2006.

[7] A. Riazanov and A. Voronkov. Splitting without Backtracking. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence* , pages 611–617. Morgan Kaufmann, 2001.

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

[9] S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.

[10] G. Steel. Visualising First-Order Proof Search. In C. Aspinall, D. Lüth, editor, *Proceedings of User Interfaces for Theorem Provers 2005*, pages 179–189, 2005.

[11] G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 406–410. Springer-Verlag, 2000.

[12] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, Submitted.

[13] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[14] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.

[15] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS Version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 275–279. Springer-Verlag, 2002.

ACL2s: "The ACL2 Sedan"

Peter C. Dillinger Panagiotis Manolios Daron Vroon

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280, USA
{peterd,manolios,vroon}@cc.gatech.edu

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
moore@cs.utexas.edu

Abstract

ACL2 is the latest inception of the Boyer-Moore theorem prover, the 2005 recipient of the ACM Software System Award. In the hands of experts it feels like a finely tuned race car, and it has been used to prove some of the most complex theorems ever proved about commercially designed systems. Unfortunately, ACL2 has a steep learning curve. Thus, novices tend have a very different experience: they crash and burn. As part of a project to make ACL2 and formal reasoning safe for the masses, we have developed ACL2s, the ACL2 sedan. ACL2s includes many features for streamlining the learning process that are not found in ACL2. In general, the goal is to develop a tool that is "self-teaching," *i.e.*, it should be possible for an undergraduate to sit down and play with it and learn how to program in ACL2 and how to reason about the programs she writes.

Keywords: ACL2, Eclipse, theorem proving, script management

1 Introduction

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." It is the name of a programming language, a first-order mathematical logic based on recursive functions, and a mechanical theorem prover for that logic [9,5,4]. ACL2 is an industrial-strength version of the Boyer-Moore theorem prover [2] and was developed by Kaufmann and Moore, with early contributions by Robert Boyer; all three developers were awarded the 2005 ACM Software System Award for their work. Of special note is its "industrial-strength": as a programming language, it executes so efficiently that formal models written in it have been used as simulation platforms for pre-fabrication requirements testing; as a theorem prover, it has been used to prove the largest and most complicated theorems ever proved about commercially designed digital artifacts.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

ACL2's power comes with a steep learning curve. This is not an issue of documentation. ACL2 has extensive documentation, including tutorials, a user's manual, workshop proceedings, and related papers, all of which are available from the ACL2 homepage [9]. ACL2 is also described in a textbook by Kaufmann, Manolios, and Moore [5], and there is also a book of case studies [4]. The sources for ACL2 are freely available on the Web, under the GNU General Public License.

ACL2's steep learning curve is due to two major factors. The first factor is usability. Beginners have to use ACL2's command line user interface and are encouraged to learn GNU Emacs. Once they start proving theorems, they are confronted with the problem of developing a mental model of what ACL2 is doing, something that is inherently difficult: reasoning about a system that reasons about other systems. Driving a user interface that is unfamiliar, non-intuitive, and happily permits lots of illogical actions distracts new users from what is important.

The second factor is the ACL2 logic. ACL2 has this tremendous advantage over many other theorem provers: it is grounded in a programming language. This makes it very easy to introduce ACL2 to users with a computer science background. However, once the logic is introduced, termination becomes an issue. In order to guarantee soundness, ACL2 only accepts functions that are shown to terminate. While ACL2 can do this automatically in many cases, there are also simple cases that require user guidance. Termination reasoning in ACL2 is very powerful because it is based on the ordinal numbers. While students and beginners eventually understand (and are even sometimes fascinated by) the ordinal numbers, their introduction significantly increases the knowledge required for interesting interaction with ACL2. Note that termination is not only used to admit recursive functions, it induces sound induction schemes, which play a central role in ACL2.

To address the above two factors and thereby make ACL2 more accessible to beginners, we have developed and released ACL2s, the ACL2 sedan [3]. ACL2s is available at `http://www.cc.gatech.edu/home/manolios/acl2s` and is being developed with the goal of making formal reasoning accessible to the masses, with an emphasis on building a tool that any undergraduate can profitably use in a short amount of time.

To address the usability factor, ACL2s features a modern graphical integrated development environment in Eclipse that provides an intuitive, robust "script management" interface with an improved front-end to the familiar "command line" interface. The prior is good for augmenting or curtailing the current theory while the latter is good for querying, testing, or debugging the current theory. ACL2s permits the user to switch between the two without fear of either one misrepresenting the relevant logical history. Other features help to eliminate other simple misunderstandings that distract from the specification and proof process: full syntax highlighting, syntax error underlining, auto-indenting, character pair matching, and input command demarcation. In addition, "session modes" serve to hide complicated functionality from novice users.

To address the logic factor, we have developed and incorporated into ACL2s *CCG termination analysis* [8]. This is a powerful, state-of-the-art termination analysis method which significantly automates termination arguments. This eliminates the need for students to justify the kind of user-defined recursive functions and in-

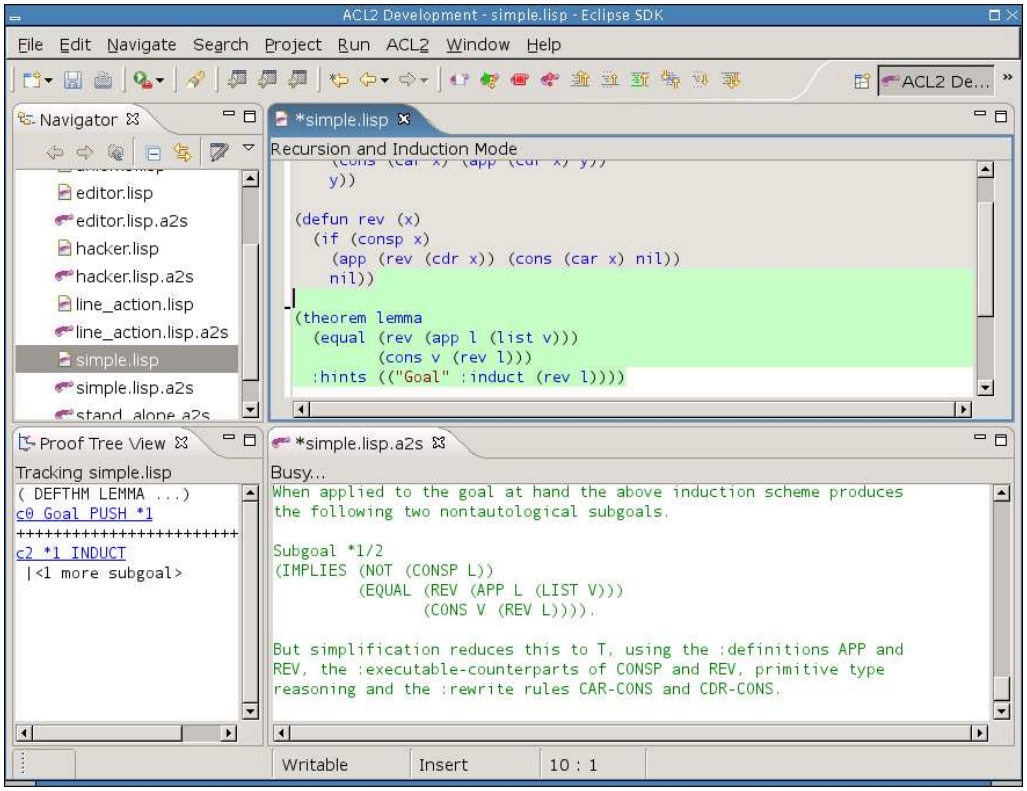


Fig. 1. **Snapshot of Eclipse workbench running ACL2s.** The top-left frame is Eclipse's file Navigator view. The top-right frame is an ACL2s source code editor. The bottom-right is an ACL2s session editor associated with that source code editor. The bottom-left is a "proof tree" view of proof happening in the session editor.

duction schemes that would be covered in an undergraduate class. We can therefore avoid discussing termination analysis completely or until well after students have become proficient ACL2 users. In addition, ACL2s includes several levels appropriate for beginners through experts. This allows us to introduce the major concepts in ACL2 in easy-to-understand modules that do not overwhelm beginners.

Together, the features of ACL2s lower barriers to learning specification and verification in ACL2—which was the sense after two graduate courses switched to requiring use of the tool.

2 GUI Overview

In the Eclipse *workbench*, development with ACL2s is centered around two types of *editors*: the source editor (also called "Lisp editor" or ".lisp editor"), and the session editor (or ".a2s editor"). In most cases, the user will use linked pairs of these editors, such as editing `somefile.lisp` and `somefile.lisp.a2s`. Based on their naming, the plugin links these so that each provides a consistent view of their shared logical *history*.

2.1 Line(s) in the Source Code

The source editor is where the user enters top-level definitions and commands as if programming offline, but the editor also provides *script management*[1] capabilities,

providing what ACL2 literature calls "the line" [5]. The editor actually maintains two lines, which we call the "completed line" and the "todo line." Because the "completed line" is never beyond the "todo line", the lines induce three (potentially empty) regions in this order: the "completed region" (gray highlight and read-only), the "todo region" (green highlight and read-only), and the "working region" (no highlight, read/write). These regions can be seen in Figure 1. The lines and other metainformation are stored in special comments on disk, preserving source code compatibility.

User interface actions grant essentially free manipulation of the "todo line", regardless of the state of the associated ACL2 session, or whether it's even running. The "todo line" will only advance past syntactically well-formed ACL2/Lisp input and only at the granularity of whole commands (see Section 4.1). Moving the "todo line" can have consequences including initiating processing of "todo" forms, interrupting the processing of a form no longer in the "todo" region and "UNDO"ing of completed commands (see Section 4.3). As ACL2 is processing forms from the "todo" region, it advances the "completed" line on success and resets the "todo line" to the point of the "completed line" on failure. If ACL2 is restarted, the "completed line" is moved to the top. In each case, we are maintaining the simple invariant that the "completed" forms have been accepted by ACL2 (and have not been undone), the "todo" forms are being processed (if the ACL2 session is running), and the "working" area is freely editable.

"Script management"-style interaction usually involves the session editor as well, which gives ACL2's output in response to forms processed as a result of line motion. The session editor shows almost exactly what the user would see if she had been manually copying processed forms into a terminal running ACL2—though the session editor has some significant enhancements for browsing output (see Section 5.3).

2.2 Command Line

Even with a script-style interface, lots of ACL2 interaction does not make sense from such an interface. We therefore made the session editor much more than a provider of detailed output. The session editor implements a command line interface to the same session used by the script-style interface of the source code editor. Most importantly, the user cannot "trick" ACL2s into an inconsistent state by switching between the two interfaces. The basic mechanism for this is copying any successful, *relevant* commands (see Section 4.2) submitted at the prompt in the session editor to the "completed region" in the associated source code editor.

The session editor looks like a dump of the input and output to a sequence of ACL2 sessions, but input coming from the "todo" region and input typed at the prompt look the same in the history.

2.3 Other UI Pieces

ACL2s also incorporates a clickable *proof tree* view, much like the proof trees provided by the Emacs interface to ACL2. Our tool takes the view a step further, though, by remembering the final proof tree of all completed commands and bringing them up as the cursor is moved to corresponding sections of the session editor.

So far, the only wizard provided by ACL2s is a "New ACL2s/Lisp file" wizard, which allows the user to pick a session mode (see Section 3.1) and has some options for generating some skeleton code that commonly appears at the top of ACL2 files.

2.4 User Experiences

Our above description of some intricate interaction between the session editor and the source code editor do not translate to difficult understanding by users. ACL2s has been a required tool for two graduate courses with an introduction to theorem proving, and students have understood our merger of the script management and command line interfaces almost immediately.

3 Language Extensions

As part of ACL2s, we have made some extensions to the underlying ACL2 tool, but we have always made sure not to disable or obscure any ACL2 functionality available outside of ACL2s.

3.1 Session Modes

Analogous to "language levels" in DrScheme [10], ACL2s offers (at present) three modes of behavior for the underlying ACL2. In teaching ACL2, the modes can be introduced in this order:

- *Programming Mode.* This mode is intended to introduce new users to ACL2 as a programming language of untyped, total functions. None of the ordinary restrictions relating to logical soundness apply. With the exception memory exhaustion (heap or stack), no runtime errors are possible with functions defined in Programming Mode. Macro definition and usage is also available in this mode.

  Implementation Note: Readers with knowledge of ACL2 will note that this is similar to the built-in "program mode" for definitions, but there is at least one important difference: runtime checking of *guards*. Guards facilitate fast, raw lisp execution but are irrelevant to the logical language of ACL2. To novices, guard checking is a distraction, which is why our Programming Mode disables it. ACL2 version 3.0 has fixed the shortcoming in "program mode" by adding an option to turn off *all* guard checking. This will eliminate the need for the hack we currently use to implement our "Programming Mode."
- *Recursion & Induction Mode.* Defining functions and macros in this mode is just like in pure ACL2, except that the theorem prover is able to prove termination of most terminating functions with no help (using CCG termination analysis, described in Section 3.2). Theorem proving in this mode is accomplished with a macro that inhibits automatic, guessed induction and adding the rules generated by the theorem to the enabled *theory*. To perform induction, therefore, the user must provide an explicit hint with the scheme to use, forcing the user to think carefully about when and how induction should be applied. Utilizing user-defined lemmas (theorems) as proof rules also requires explicit hints. This helps users to focus on individual proofs rather than building a coherent theory, which is harder still.

```
(defun sum-lists (x y)
  (if (or (consp x) (consp y))
      (cons (+ (car x) (car y))
            (sum-lists (cdr x) (cdr y)))
    nil))
```

Fig. 2. The function, `sum-lists`, takes two lists and returns the list whose elements are the sums of the elements of the inputs. If one input list is longer than the other, the returned list is as long as the longer and the smaller is considered to be padded with 0s.

- *Pure ACL2* This mode is just like regular ACL2, except that we offer CCG termination analysis as a convenience.

3.2 CCG Termination Analysis

Termination in ACL2. A significant stumbling block for new users and a source of frustration for experienced users of ACL2 is termination. Every function admitted to ACL2 must be proven to terminate for all inputs before ACL2 will accept it. First, this guarantees the logical consistency of function definitions—that every syntactically legal function application corresponds to exactly one value. Second, ACL2 derives induction schemes from recursive functions, and those induction schemes are sound as a consequence of termination of the corresponding function. Induction is an integral part of the theorem proving capability of ACL2, especially in proving properties over infinite classes of input.

To prove termination of a function, ACL2 uses a specified or guessed *measure* to map the function's inputs into values in the domain of a *well-founded* relation, such as the $<$ relation on the natural numbers. If the measure always returns a value in the relation's domain and recursive calls always use inputs that, according to the measure and well-founded relation, are "smaller than" the previous, it cannot go on forever. (No sequence decreasing according to a well-founded relation can be infinite.)

ACL2 uses only simple heuristics to guess measures when proving termination, so it is easy to define functions for which ACL2 is not able to guess the correct measure. Therefore, new users soon find the need to learn about engineering and justifying measures to ACL2, which tends to overwhelm those who are still struggling to prove simple theorems.

For example, ACL2 cannot guess the measure for the `sum-lists` in Figure 2. Using $<$ (the normal less-than relation) over the natural numbers as our well-founded relation, our measure can be `(+ (len x) (len y))`, where `len` returns the length of a list (0 for atoms, and $1 +$ `(len (cdr x))` for `cons`es, `x`).

Mechanics of CCG analysis. In [8], we introduce a new, more automatic termination analysis based on CCGs, or *context calling graphs*. Within a function (or set of mutually recursive functions) we augment each recursive call site with predicates that are needed to get there ("*rulers*") within the function. These augmented call sites are *calling contexts*, and a calling context graph (CCG) is a graph whose vertices are calling contexts and has an edge from e to e' if e calls the function con-

taining e' and e can lead directly to e' during execution. Intuitively, a CCG is like a call graph but in terms of call sites instead of functions, giving it finer granularity.

Now we apply the notions of *measure* and *well-founded relation* CCGs. Suppose we can assign a set of measures—called *calling context measures*, or CCMs—to each context such that every infinite path through the CCG has a corresponding sequence of CCMs that never increase and decrease infinitely often. It follows that every computation must then terminate. Theorem proving plays a key role in this analysis as it is used to prune edges from CCGs and to determine when CCMs are non-increasing or decreasing as we traverse edges in CCGs.

Our algorithm uses heuristics to pick CCMs, together with a sufficient condition for the above path-related criterion that is based on [7]. More details and other improvements are described in our extended abstract [8].

Results of using CCG analysis. We ran our CCG implementation on ACL2's regression suite. This is a collection of ACL2 libraries on a variety of topics including arithmetic, set theory, processor verification, and model checking. These libraries were submitted by various members of the ACL2 community over a decade, and are therefore representative of typical ACL2 usage. The regression suite contains over 10,000 function definitions, a significant number of which required explicit user intervention to prove termination. When running our termination analysis on the regression suite, we discarded explicit user hints, and provided no manual assistance. Our analysis successfully proved 98.7% of the 10,000 functions terminating, including 68.2% of those that previously required explicit user hints.

We have implemented our algorithm into the current version of ACL2s [6]. The result is that ACL2s now proves termination automatically for a much higher proportion of functions, particularly among simpler functions that new users tend to define. The `sum-lists` function, for example, is easily proven to terminate by our analysis. With our analysis, a discussion of the complex concepts of termination analysis can be postponed for new users, allowing them to become more familiar and comfortable with the basic concepts of ACL2 first. In addition, advanced users can spend less time carefully engineering and justifying measures.

4 Script Management

Implementing a powerful, robust "script management"-style interface for ACL2 was non-trivial. First, we would need to be able to detect entire input forms for ACL2. Next, some input forms require explicit "undo" while others have no effect other than printing some result. Others still are not undoable with the regular "undo" command, but we do not want to restrict the commands available from the interface.

Another complication has to do with our command line interface. Recall that successful, "relevant" commands entered at the session editor's command prompt are inserted at the completed line in the source editor. We needed to come up with a notion of "relevance" that made sense.

4.1 Input Demarcation

To know where to move the todo line when the user asks to advance it one ACL2 form, we implemented a Common Lisp parser in Java. This parser (call it the "batch parser") was implemented before and independent of the parser that does syntax highlighting and checking for the editors (call that one the "online parser"; see Section 5.1). The batch parser also pulls entire ACL2 forms typed at the session editor's command prompt.

Another view of the job of the batch parser is to make sure that each time the ACL2 reader asks for an expression, it is given exactly one syntactically well-formed expression. "Well-formed" in this case means that it will not generate a *read error* by ACL2's reader. In the case of ill-formed input, the batch parser generates an appropriate error message with a relevant location in the input. With the batch parser in place, neither the plugin nor the user need worry about odd behavior from ACL2 in recovering from read errors or getting stuck with ACL2 expecting more input but not know exactly what is needed to complete an expression. These cases are particularly frustrating to novices.

ACL2's *keyword commands* are convenient, but are more prone to that "what else am I supposed to type" experience when used at a terminal because they potentially require several expressions to compose a single input form. Our tool imposes a stricter interpretation of keyword commands that, in a sense, fixes the confusion: keyword commands are terminated *only* by a newline outside of a Lisp expression, which corresponds to existing conventional usage. The mechanism enabling us to adopt this interpretation is our plugin's translation of keyword commands to their non-keyword equivalents before giving them to ACL2. This means that if the wrong number of parameters is given, instead of ACL2 blocking or misinterpreting input, it simply reports, "wrong number of arguments."

4.2 Input Classification

The next step in our solution was to classify each input form before letting the underlying session execute it. This would tell us (ACL2s) about the form's relevance and also provide some useful feedback to the user. In fact, the only real textual difference between a terminal dump and our session editor contents is that we prepend each input with a classification that categorizes the potential effects it could have. Figure 3 shows some examples.

Classifications can depend on the history of the particular session, so the easiest way to classify an input is inside of ACL2. Part of our extensions to the ACL2 core is some code for classifying inputs, that builds on some existing code in ACL2 for deconstructing and analyzing inputs. Also, we designed our Eclipse plugin code that interfaces with the ACL2 core to be able to handle requests that are hidden from the user. The output seen in the session editor is *not* all the output that the ACL2 session has generated; it's only the output that is relevant to the forms entered by the user!

Here are most of the possible classifications:

- *EVENT* is usually a definition of a function or theorem and is always considered relevant because it could modify or extend the logical *world*. Events also play an

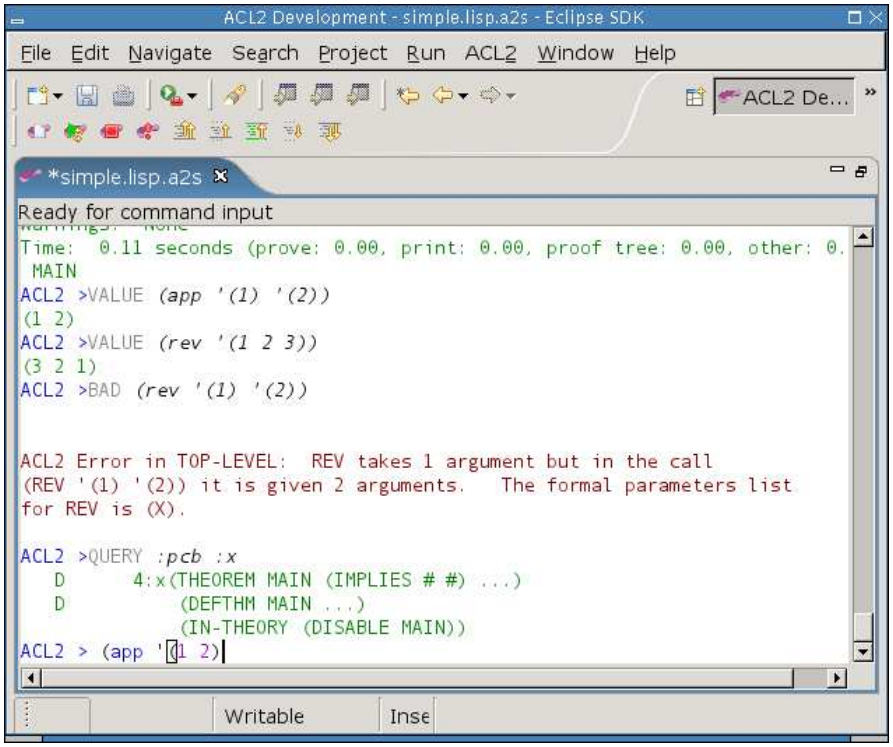


Fig. 3. **Close-up of the Session Editor.**

important role in book development, discussed in Section 5.2.

- *VALUE* is some computation that, by the top-level function's signature, is unable to change any *state*, including the logical world. The best example is testing a function on some input to see its result. Values are irrelevant.
- *QUERY* is one of many built-in commands that relate to the logical world, but are known not to change anything. Examples include printing the rules associated with a symbol or trying to prove an unnamed (thus, immediately forgotten) conjecture. Queries are irrelevant.
- *BAD* is given to an input if the categorization code is able to detect an error the plugin's batch parser is not. Such semantic errors include trying to invoke a function with the wrong number of parameters. Because BAD inputs always fail, they are, in a sense, inherently irrelevant.
- *COMMAND* changes something in ACL2 that the built-in undo mechanism does not handle but ACL2s can undo cleanly and reliably. To be safe, commands are considered relevant.
- *IN-PACKAGE* is a special COMMAND that changes the current package. It is, thus, relevant, but is singled out for its role in book development (see Section 5.2).
- *ACTION* is a catch-all for inputs that could have effects we don't know how to undo. It would be rare for a novice to invoke such an input and rarer still for a non-undoable effect to affect soundness. Actions are considered relevant and cause a warning to be printed when they are undone (to the extent we are able).
- *UNDO* and *REDO* are generated by motion of the "completed" line, as

discussed below.

These classifications give us a sane way of implementing script management and maintaining consistency between the two editors.

4.3 Undo

To have clean, powerful support for retreat of the "completed" line, the modified ACL2 core used by ACL2s includes an undo mechanism that is, in a sense, more primitive than the built-in undo mechanism. In other words, an ACL2 undo can be used as just another command on top of ACL2s's undo mechanism. Basically, our mechanism keeps a list of old pseudo-states, each of which contains a logical world and many other settings that affect the treatment of input. To perform an UNDO, the Eclipse plugin simply invokes the mechanism for restoring a previous pseudo-state. It is non-trivial for the ACL2s user to invoke this mechanism with a command (rather than causing the plugin to invoke it in response to line motion) because using the mechanism requires knowledge of a secret number chosen at random for each session. The secret is hidden in the ACL2 session such that only an expert who *really* wanted to usurp our undo structure would be able to do so.

4.4 Redo

Whenever ACL2s performs an UNDO, the pseudo-state it was in before the UNDO is not forgotten—nor is any pseudo-state that got us here by some sequence of UNDOs, REDOs, and irrelevant forms. Thus, we have a mechanism to return to such states. A REDO is actually invoked if, following the UNDO of a form x plus any sequence of matching UNDO/REDOs and irrelevant forms, a form y is submitted with the same abstract syntax as x. Two forms have the same abstract syntax if they parse to the same Lisp objects. Comments, for example, are irrelevant to abstract syntax, but an extreme example would be

```
(1+ 42) ≅ (Ac|L|2::1\+ #c(840/20 -0.) . ())
```

A nice consequence of the REDO mechanism is the ability to modify comments and such above the line in a way that ensures ACL2 would process it the same way, but without having to wait for ACL2 to reprocess it. Simply retreat the line high enough, make the modifications, and move the line back to where it was. If the abstract syntax is unchanged, the completed line will move back to where it was almost immediately using REDOs. In fact, ACL2s even allows this with no session running! (It pretends to perform the UNDOs and REDOs that would be legal.)

5 Editor Features

5.1 Source Code

The source code editor can be used to edit .lisp files even when no corresponding .lisp.a2s file is present, meaning the editor is not paired with a session editor. The editor complains about some Common Lisp syntax that is illegal in ACL2, but an option to disable those cases might be included in future releases.

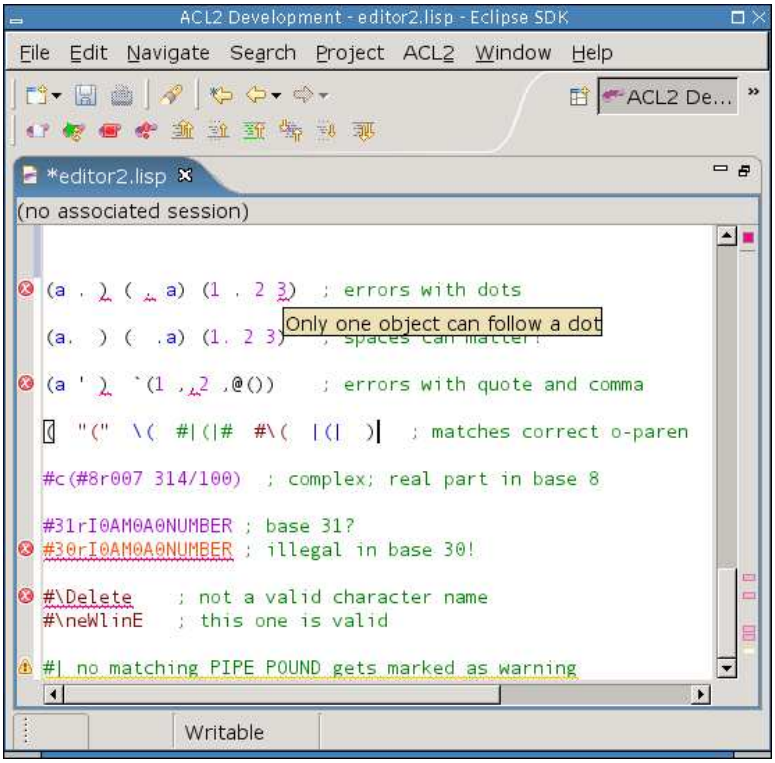


Fig. 4. **Close-up of the Source Editor.**

An important part of the source editor is the "online parser" it utilizes, which is a hand-coded incremental Lisp lexer with some parser-like capabilities as well. This part of the plugin is responsible for dividing up and classifying tokens for syntax-based coloring, depicted in Figure 4. It also computes matching character pairs and annotates the code in the case of any illegal syntax. The token representation also plays a role in intelligent auto-indenting.

The editor matches open and close parentheses, as one would expect, but it also matches double-quotes around strings, pipes within symbols, potentially nested `#|`-style comments, and parentheses within comments. Particularly impressive to a crowd of ACL2 experts was the editor's matching of "`,`" and "`,@`" tokens to their respective backquote characters.

Only one type of potential ACL2 read error is not annotated in the source editor: the undefined package error. The set of defined packages can grow dynamically, so such errors are not identified until checked by the batch parser. All other errors show up as usual in Eclipse, with red or yellow underlining, a mark in the *overview ruler*, and a message that appears when hovered over. Examples are depicted in Figure 4.

The auto-indenting is much like Emacs or DrScheme. A notable, much-praised exception is indenting inside of string literals according to rules followed by ACL2's built-in functions for formatted output.

The editor scales nicely, for it performs acceptably fast on (ASCII text) Lisp files from the ACL2 source code that exceed a megabyte. The only exception is when a change causes the reformatting of almost the entire megabyte file—such as

commenting out the whole file. This can take a few seconds to complete, but the bulk of that cost is Eclipse applying off-screen style information we give it.

5.2 Book Development

In many cases, ACL2 development is the development of a *book*, which is basically a reusable collection of definitions. Defining a book, though, can be tricky for several reasons, most of them relating to the requirement that books be processable by Common Lisp outside of ACL2. First, the *preamble* must be processed by ACL2 before the contents of the book, but it cannot simply appear above the book in the book's source file. Some users put the preamble into a special comment, some put it in a separate file, and many do both. This is a pain. In ACL2s, the preamble can be written directly at the top of the source file, though on disk it is stored in a comment (and possibly another generated file).

ACL2s has its own special construct for marking the end of the preamble: `(begin-book)`, which actually takes some optional parameters that are given to `(certify-book ...)` when the user asks to *certify* the book. When submitted to a running session, `(begin-book)` does not do anything special—nor does it need to.

What does happen when `(begin-book)` is submitted, is the source editor begins highlighting that part of the completed region with light purple instead of gray. As more forms are completed, the light purple highlight extends as long as the forms are legal for a book. After the `(begin-book)` this must be an `(in-package ...)`. After that, only EVENTS (see Section 4.2) are legal.

During book development, it is not unusual for the user to move the line past some forms that are not legal within books. This is fine, and we call this a "tangent". When the user is ready to undo his tangent, the light purple highlighting indicates the point where legal book constructs were last abandoned.

We have not yet implemented "one click" certification of books within ACL2s, but the infrastructure is mostly there.

Finally, if `(begin-book)` is never used, no preamble is stored and no special highlighting of the completed region is done. The book development features do not complicate things if not utilized.

5.3 Session Editor

The session editor is the locus of our improved command line interface to ACL2 and captures much more than just a "dump" of the input and output. The saveable and restorable session history is a sequence of $\langle \mathit{Environment}, \mathit{Input}, \mathit{Output}, \mathit{Status} \rangle$ tuples. The *Environment* contains information such as the current Lisp package, the length of the logical history, the prompt printed to the user, and other settings that can influence the meaning of input. The *Input* captures the concrete and abstract syntax of an input form and a categorization describing its potential effects. The *Output* stores the text of the output, the location of *checkpoints* within the output, and the final *proof tree* associated with the command. The *Status* indicates whether the command was successful and, if not, whether it was interrupted.

The editor uses color to distinguish sources or types of text. For example, the

prompt is blue, the input categorization is gray, the actual input is black, and the output is red on failure or green on success, as depicted in Figure 4.

The output can be navigated like any other read-only editor in Eclipse, but there are special shortcuts for traversing from input to input and among checkpoints within a single command's output.

The only editable region is beyond the final prompt, and it is only editable if ACL2 is waiting for input. This region, which we call the "immediate" region (for typing "immediate" commands), uses the same online parser and presentation scheme as the source editor. The command line interface, therefore, has character matching, syntax error highlighting, and even auto-indenting. The history of immediate commands is also navigable, much like in a UNIX shell.

The typed immediate command is submitted when *Enter* is pressed, though it's not quite that simple. When *Enter* is pressed, the batch parser checks to see if some prefix of the typed input is a syntactically well-formed input form. If not, the *Enter* simply causes a newline to be inserted. If a prefix is a full command, that prefix is removed and submitted as the next input. This could leave some text left over if an eager user decides to type more than one command at a time. The immediate text disappears while ACL2 is busy but reappears once another command is expected.

The session editor also supports typing input to ACL2 that is not command input. For example, ACL2 sometimes prompts the user for an answer to a yes/no type question. Another example is interacting with the *proof checker*, which has its own set of commands. All of these cases expect a single Lisp object as input, and our plugin is able to detect when ACL2 is expecting non-command input. Non-command input is never taken from the source editor, but must be typed in the session editor, which, in fact, tracks an independent command line history for non-command input.

One never interacts with raw Lisp from ACL2s. There are cases in ACL2 in which errors take the user to a raw Lisp prompt and the user must manually break out of it to return to ACL2. ACL2s takes care of this for the user, partly for convenience but partly because it's hard to determine when raw Lisp is expecting input.

6 Related Work

6.1 ACL2 in DrScheme

Researchers at Northeastern University have hooked ACL2 into DrScheme [12]. Their preliminary system has some features we would like to have in ACL2s, but it also has some inherent limitations.

One part of the system is a DrScheme language for "ACL2 Beginner," which is an attempt to duplicate the ACL2 language using Scheme macros and Scheme functions. This simulated ACL2 benefits from the features of the DrScheme development environment, including its simple GUI, its debugging features, and its static checking features. The feature that would be most difficult to mimic is the "Check Syntax" feature, which performs macro expansion in a way that allows uses of lambda variables to be linked to their point of definition/declaration in the

original source code.

The complication, however, is that Scheme is only partially compatible with Common Lisp, the basis of ACL2 proper. A clean embedding of full ACL2 in Scheme probably is not possible, due to incompatibilities such as packages and other namespace issues. The current "ACL2 beginner" language has other incompatibilities, including use of functions as first-class values and a restricted macro language.

Not necessarily an incompatibility but, in our opinion, a poor design decision for the current "ACL2 beginner" language was incorporation of *contracts* for functions, analogous to ACL2 *guards*. Guards are not part of the logical ACL2 language, so we feel such dynamic type errors complicate a beginner's ACL2.

ACL2 in DrScheme also provides a basic script management interface for interacting with the theorem prover. The implementation is still rough and easy to break, but of theoretical concern is the relative independence of the two interfaces. One can track two separate logical environments that pertain to the same input buffer. For example, defining a function in the *read-eval-print* interface does not cause it to be defined in the theorem prover.

6.2 PG/Eclipse

The Eclipse version of Proof General has some nice features [13,11]. The project is ahead of ACL2s in terms of utilizing the graphical interface for browsing help, documentation, and the logical world. As we have overcome the technical challenges of hooking a robust script-management interface to ACL2, someone hooking ACL2 to PG/Eclipse using the *Proof General Interaction Protocol* could utilize our extensions to the ACL2 core.

7 Conclusion

In this paper, we have described ACL2s, the ACL2 sedan. ACL2s is a publicly available system that we have developed in order to make formal reasoning more accessible to the masses. One of our goals is to create a "self-teaching" system that enables undergraduates to learn how to prove theorems about the computing systems they design by "playing" with ACL2s. As an initial step in this direction, ACL2s includes many novel features for streamlining the learning process. This includes a modern graphical integrated development environment in Eclipse that provides an intuitive, robust "script management" interface with an improved front-end to the familiar "command line" interface. It also includes CCG termination analysis, a powerful, state-of-the-art termination analysis method which essentially eliminates the need for students to justify the kind of user-defined recursive functions and induction schemes that would be covered in an undergraduate class. Together, the features of ACL2s lower barriers to learning specification and verification in ACL2—which was the sense after two graduate courses switched to requiring use of the tool.

For future work, we are planning to use ACL2s to teach an undergraduate course on processor design. The goal is that students should learn more about processor

design than they would have learned without the use of ACL2s. We plan to accomplish this by having student use ACL2s as a specification language and as an oracle that will be configured with the use of various libraries we are developing to either prove the correctness of the student designs or to provide useful information for finding errors. We also want to extend ACL2s so that it can provide more visualization and query support for proving theorems.

Acknowledgments

We would like to thank Matt Kaufmann for help and guidance in hacking ACL2, Qiang Zhang for bug hunting, and Alexander Spiridonov for providing ACL2s support to students at the University of Texas at Austin.

References

[1] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. Journal of Symbolic Computation, 25(2):161–194, 1998.

[2] Robert Stephen Boyer and J Strother Moore. A Computational Logic Handbook. Academic Press, second edition, 1997.

[3] Peter C. Dillinger, Panagiotis Manolios, J Strother Moore, and Daron Vroon. ACL2s, the ACL2 sedan. `http://www.cc.gatech.edu/~manolios/acl2s`.

[4] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, June 2000.

[5] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, July 2000.

[6] Matt Kaufmann, Panagiotis Manolios, J Strother Moore, and Daron Vroon. Integrating CCG analysis into ACL2. In Eighth International Workshop on Termination, August 2006. Part of FLOC '06.

[7] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In ACM Symposium on Principles of Programming Languages, volume 28, pages 81–92. ACM Press, 2001.

[8] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In Thomas Ball and Robert Jones, editors, Computer-aided Verification (CAV) 2006, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[9] J Strother Moore and Matt Kaufmann. The ACL2 home page. `http://www.cs.utexas.edu/-users/moore/acl2/`.

[10] PLT Scheme. DrScheme, 2003. See URL `http://www.drscheme.org/`.

[11] Proof General Project. PG/Eclipse, 2006. `http://proofgeneral.inf.ed.ac.uk/eclipsewiki`.

[12] Dale Vaillancourt, Rex Page, and Matthias Felleisen. ACL2 in DrScheme, 2006. `http://www.ccs.neu.edu/home/dalev/acl2-drscheme/`.

[13] D. Winterstein, D. Aspinall, and C. Lüth. Proof General / Eclipse: A generic interface for interactive proof. In User Interfaces for Theorem Provers. ENTCS, 2005.

Enhancing Theorem Prover Interfaces with Program Slice Information

Louise A. Dennis[1,2]

School of Computer Science and Information Technology
University of Nottingham, Nottingham, UK

Abstract

This paper proposes an extension to theorem proving interfaces for use with proof-directed debugging and other disproof-based applications. The extension is based around tracking a user-identified set of rules to create an informative program slice. Information is collected based on the involvement of these rules in both successful and unsuccessful proof branches. This provides a heuristic score for making judgements about the correctness of any rule.
A simple mechanism for syntax highlighting based on such information is proposed and a small case study presented illustrating its operation. No implementation of these ideas yet exists.

Keywords: Proof-Directed Debugging, Program Slicing, Verification

1 Introduction

The use of verification for locating errors in theorems, and more specifically programs, is a relatively neglected area as is the provision of interfaces to assist in this task. This paper considers the proof-directed debugging of functional programs and proposes an extension to current theorem proving interfaces to support this.

The extension is based on the assumption that the debugging process involves locating a program statement or, in the case of functional programs, function case which is incorrect. This incorrect statement will appear in a *program slice* which can be identified during verification. Other program slices leading to correct deductions may also be identified during proof. This information can then be used to create appropriate syntax highlighting of function cases in an interface. A potential highlighting scheme is put forward and a simple case study based around Isabelle/HOL [12] and ProofGeneral [1] is performed to show how this would work.

No implementation has yet been performed however potential issues are discussed in the context of Isabelle Proof General.

[1] This research was funded by EPSRC grant GR/S01771/01 and Nottingham NLF grant 3051.
[2] Email:lad@cs.nott.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Although the discussion in this paper is based around an application to proof-directed debugging it is likely that similar mechanisms may also be useful in other situations where the cause of a proof failure needs to be identified.

The paper is organised as follows: §2 discusses the concepts of proof-directed debugging and program slicing; §3 present a mechanism for tracking program slices through a proof and §4 presents examples of this mechanism at work via a simple case study; §5 discusses some results using a similar mechanism within an automated system; §6 looks at some related work and §7 discusses implementation issues and other further work.

2 Proof Directed Debugging and Program Slicing

Proof-directed debugging was first suggested by Harper [10] and work is underway to extend this into a framework for locating program errors through the proof process [6]. The idea of using a framework rather than relying on a user's skill at general proof, is based on the example of Algorithmic debugging [15,9,11]. Algorithmic debugging constructs an execution tree of a run of the program on some input and then queries the user each time this tree branches. This identifies branches which are returning false results and so locates sections of code responsible for errors.

Program Slicing was first suggested by Weiser [17]. The key idea was to identify a variable of interest at some point in a program (called the *slicing criterion*) and then extract a fragment of the program (a *program slice*) either containing all those statements upon which the value of the variable at that point depended or that fragment whose values were effected by the value of that variable at that point. Program Slicing techniques for imperative languages have generally followed this work [16] using control flow graphs, data flow graphs or other graph-based representations of programs with statements represented as nodes in the graph and a program slice as a set of nodes from the graph. In functional programs function application takes the place of program statements. The notion of a slicing criterion can also be generalised (e.g. to a projection as in [14]).

The intention behind proof-directed debugging is to use the branching structure of a proof to create program slices and use these to assist in the location of errors. There is clearly a need to provide appropriate tools (i.e. tactics/Isar methods) tailored to this task. This paper does not concentrate on this aspect but considers instead the way a theorem prover's interface could assist a user through the presentation of relevant program slices.

3 Proof Tree Branches as a Slicing Criterion

The verification of functional programs naturally involves splitting a program into a set of equational rules each corresponding to a case in its functional definition. The usage of these rules in the proof can thus be tracked, effectively creating a program slice (ie. those parts of the program used in the proof of any program), and a "score" maintained indicating how many true and false branches of the proof have used that rule (typically as part of a simplification process). These scores can then be used by the interface to return additional information to the user. For simplicity,

we shall continue to refer to these rule traces as program slices even though, in the context of theorem proving, there is no reason why they should not be a general collection of definitions, lemmas and theorems unrelated to a any program.

Let us consider a simple insertion sort program written in ML.

```
fun insert x [] = [x]
  | insert x(h::t) =
          if x ≤ h then x :: h :: t
                   else h :: insert x t;

fun sort [] = []
  | sort (x :: xs) = insert x (sort xs);
```

Each case of the definition becomes one of four equational rules:

(i) `insert x [] = [x]`

(ii) `insert x (h::t) = if x ≤h then x :: h :: t`

`else h :: insert x t`

(iii) `sort [] = []`

(iv) `sort (x :: xs) = insert x (sort xs)`

We suggest that a proof-directed debugging interface should allow a user to nominate a selection of such definitions as "suspect" during a verification attempt. Obviously a user could choose to nominate all definitions involved in their development as suspect including ones related to the specification and even pre-existing definitions from the theorem prover's theory database however our suspicion is that this would lead to an overloading of information rendering program slicing of little use. This is an obvious subject for some experimental investigation once such an interface has been implemented.

We assume that a theorem proving system generates a sequence of *proof states* which, at the very least, contain lists of current open goals in the proof attempt. The central idea is to associate program slices with the goals in these proof states. Each goal, g, in a proof state is associated with a set of suspect rules (*a slice*), $\mathcal{S}(g)$, which have been used in the derivation of that goal. In addition to this the system also stores a set of triples in each proof state, s, associating each suspect rule, r, with two integers the first of which, the good integer, $good(r, s)$ is incremented whenever a proof branch is closed (because it has been successfully proved) and the second of which, the bad integer, $bad(r, s)$, is incremented whenever a goal is derived with a False conclusion (this can be revised if a contradiction is subsequently found in the hypotheses). Where it is obvious, the state argument will be dropped from these functions. These two scores can be used to form a probabilistic estimate of the chance that a rule is correct.

Initially $good(r)$ and $bad(r)$ are set to zero for all rules, r, and the initial goal, g_i, is associated with the empty program slice, $\mathcal{S}(g_i) = []$. As the proof progresses the system updates the information as follows:

Consider two proof states, s_n followed by s_{n+1}. s_{n+1} is derived from s_n by a tactic t which replaces some parent goals with a set of child subgoals. For each new subgoal, g, in such a proof state with parent, g_p.

- Let R be the set of suspect rules used by t to derive g from g_p, $\mathcal{S}(g) = \mathcal{S}(g_p) \cup R$.
- If g has a False conclusion and g_p did not then for all rules r in $\mathcal{S}(g)$, $bad(r, s_{n+1}) =$

$bad(r, s_n) + 1$.

- If g has a True conclusion then for all rules r in $\mathcal{S}(g)$, $good(r, s_{n+1}) = good(r, s_n) + 1$. Furthermore if the conclusion of g_p was False then $bad(r, s_{n+1}) = bad(r, s_n) - 1$.
  This last modification allows a False goal to become closed (by discovering a contradiction in the hypotheses) and then corrects the bad integer to cancel out the effect produced by the previous deduction of False.
- For all remaining rules, r, $good(r, s_{n+1}) = good(r, s_n)$ and $bad(r, s_{n+1}) = bad(r, s_n)$.

On the whole it would appear to be preferable if interfaces take on the task of tracking rule usage information rather than the underlying theorem prover since this information is extra-logical. However in automated, or semi-automated systems such as proof planners (e.g. IsaPlanner [8] and $\lambda Clam$ [13]) there would appear to be benefits in tracking such information in the proof system itself so that it can inform an automated debugging process [5].

The obvious mechanism for presenting this tracking information to a user is as a syntax highlighted list of rules associated with each goal. For instance this paper will use the monochrome conventions shown in Table 1. The categories have

	Highlighting convention
$r \in \mathcal{S}(g)$	**bold**
$bad(r) > good(r)$	underline
$bad(r) < good(r)$	*italics*

Table 1
Highlighting Conventions used in this Paper

been selected because they proved to be the most informative in the examples discussed below. The are interpreted as **used to derive this goal**, probably incorrect and *probably correct*. There is no reason, in principle, why such highlighting should be restricted to just three categories. Indeed, following results in an automated system, we argue in §5 for a further category of "worst" rule based on an ordering of tuples of bad and good integers.

4 Case Study

We now show some examples of proof attempts of incorrect theorems undertaken in the Isabelle/Isar system [12,18]. These examples are drawn from a corpus of buggy student ML programs [7].

We will consider the verification of the ML program shown in figure 1. This is a real example submitted by a student as the solution to an exercise to provide a function, `onceOnly`, that when applied to a list, `l`, returned a new list containing only one copy of each element in `l`. There are three errors in this program. Firstly the basis case of the `insert` function is incorrect. Secondly a case is missing in the definition of the `Once` function (the case for lists of length one) and lastly in the `else` branch of the recursive case the expression should be `x1 :: Once (x2 :: xs)`.

An Isabelle formalisation of the student's program taken from [7] is shown in

```
fun insert x [] = []
  | insert x(h::t) =
          if x ≤ h then x :: h :: t
                   else h :: insert x t;

fun sort [] = []
  | sort (x :: xs) = insert x (sort xs);

fun Once [] = []
  | Once (x1 :: x2 :: xs) =
       if x1 = x2 then Once (x2 :: xs)
                  else x1 :: x2 :: Once xs;
fun onceOnly [] = []
  | onceOnly (x :: xs) = Once (sort (x :: xs));
```

Fig. 1. A Buggy ML Program

```
primrec
   insert_nil: "insert x  [] = []"
   insert_cons: "insert x (h#t) = (if x ≤ h then
                                   x#h#t else h#insert x t)"

primrec
   sort_nil: "sort [] = []"
   sort_cons: "sort (x#xs) = insert x (sort xs)"

recdef Once "measure length"
   once_nil: "Once [] = []"
   once_cons: "Once (x1#x2#xs) = (if x1=x2 then
                           Once (x2#xs) else x1#x2#Once xs)"

primrec
   onceOnly_nil: "onceOnly [] = []"
   onceOnly_cons: "onceOnly (x#xs) = Once (sort (x#xs))"
```

Fig. 2. Isabelle Formalisation of the Buggy program

figure 2. It should be noted that this represents a naive shallow embedding of ML into Isabelle but one sufficient for proof-directed debugging at this scale. In order to verify this program a further function, `count_list` which counts the number of occurrences of its first argument in its second was used. The first theorem to be proved is:

$$\neg x \in l \Longrightarrow count\_list\ x\ (onceOnly\ l) = 0$$

For the purposes of this case study we assume that the definitions of `insert`, `sort`, `Once` and `onceOnly` are all considered suspect which gives us eight suspect rules: `insert_nil`, `insert_cons`, `sort_nil`, `sort_cons`, `once_nil`, `once_cons`, `onceOnly_nil` and `onceOnly_cons`. We also assume that the following theorem has been proved:

$$onceOnly\ l = Once(sort\ l) \tag{1}$$

The remainder of this section is organised as follows. §4.1 illustrates slice creation in the initial stages of the proof in order to give an idea of how the information updating works, §4.2 illustrates the effect of reaching a false goal, and §4.3 illustrates what happens when cases are missing.

4.1 Basic Usage

The following table shows the information held in the initial proof state

Rule	good	bad	Rule	good	bad
insert_nil	0	0	Once_nil	0	0
insert_cons	0	0	Once_cons	0	0
sort_nil	0	0	onceOnly_nil	0	0
sort_cons	0	0	onceOnly_cons	0	0

From now on we will omit the full table but concentrate instead on the summary of the information that can be provided with syntax highlighting.

At the start of the proof there is one Isabelle goal

```
1. ¬ x ∈ l ⟹ count_list x (onceOnly l) = 0
```

to which is attached the empty slice. Presentationally it seems advisable to omit any rules defining constants not appearing in the current goal So the initial goal would display the additional information (NB. at present these rules do not fit into any of the categories described in Table 1 therefore neither is highlighted in any way):

- `"onceOnly [] = []"`
- `"onceOnly (x#xs) = Once (sort (x#xs))"`

The proof attempt proceeds by simplifying, replacing `onceOnly l` with `Once (sort l)`, according to (1), and then applying length induction on the list [3]. Since (1) isn't in our suspect list its use in simplification isn't recorded. We don't chain rule tracking back through additional lemmas so there is no record that, even implicitly, `onceOnly_nil` and `onceOnly_cons` were involved in the goal. Once again it will need experimentation with an implementation to determine whether this is a sensible choice. This gives us the following Isabelle goal:

```
1. !!xs. [| ∀ ys. length ys < length xs → ¬ x ∈ ys
                      → count_list x (Once (sort ys)) = 0;
            ¬ x ∈ xs |]
         ⟹ count_list x (Once (sort xs)) = 0
```

This introduces two new suspect constants but has so far used none of our rules. Furthermore the constant `onceOnly` is no longer mentioned and so its definitional rules are dropped from the display list. Hence the following suggested output.

- `"sort [] = []"`
- `"sort (x#xs) = insert x (sort xs)"`
- `"Once [] = []"`
- `"Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)`

  `else x1#x2#Once xs)"`

The next step is a case split on `xs` using the Isar `cases` method followed immediately by simplification of all goals. This automatically discharges the first goal associated with the case split (for $xs = []$) leaving us with one goal:

[3] It takes some experience with these styles of proof to select length induction as the appropriate scheme. At present this work presumes a user with relatively sophisticated theorem proving ability yet paradoxically rather naive program debugging skills – providing further support in the choice of Isar methods is left to further work.

```
1. !!a list. xs = a # list ⟹
          count_list x (Once (insert a (sort list))) = 0
```

Discharging the first goal creates a slice consisting of `sort_nil` and `once_nil` and updates the good integers so that $good(sort\_nil) = good(once\_nil) = 1$. The remaining goal was generated using the rule `sort_cons` and so its slice is `[sort_cons]`.

Following the syntax highlighting conventions, therefore, we get the following rule annotations:

- `"insert x [] = []"`
- `"insert x (h#t) = (if x < h then x#h#t else h#insert x t)"`
- *`"sort [] = []"`*
- **`"sort (x#xs) = insert x (sort xs)"`**
- *`"Once [] = []"`*
- `"Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)`
  `else x1#x2#Once xs)"`

already we are seeing information about program slices in which we can have some confidence and we get some information on the slice which is relevant to the current goal.

4.2 Inferring False

It becomes clear, while attempting the above proof, that some independent lemmas need to be established about the `sort` function. This provides a good example of how the system behaves when a goal evaluates to False. Let us consider a simple lemma to show that all members of a list, `l`, are also members of `sort l`.

We start with the goal:

```
theorem "x ∈ l ⟹ x ∈ (sort l)"
```

Following our previous rules and guidelines the displayed rules are:

- `"sort [] = []"`
- `"sort (x#xs) = insert x (sort xs)"`

The proof continues by length induction on `l` (which does not change the annotation) followed by a case split on `xs` and simplification of all goals. The first subgoal is discharged automatically, leaving:

```
1. !!a list.
      [| if a = x then True else x ∈ list; xs = a # list;
         ∀ ys. length ys < Suc (length list) →
                                  x ∈ ys → x ∈ sort ys;
         if a = x then True else x ∈ list |]
      ⟹ x ∈ insert a (sort list)
```

and the highlighted rules:

- `"insert x [] = []"`
- `"insert x (h#t) = (if x ≤h then x#h#t else h#insert x t)"`
- *`"sort [] = []"`*
- **`"sort (x#xs) = insert x (sort xs)"`**

It then proceeds by cases on `(sort list)` followed by simplification which gives two subgoals with their associated program slices:

```
1. [| ¬ x ∈ list; if a = x then True else x ∈ list;
      sort list = [];
      if a = x then True else x ∈ list |]
   ⟹ False
```

- **"insert x [] = []"**
- "insert x (h#t) = (if x ≤h then x#h#t else h#insert x t)"
- *"sort [] = []"*
- **"sort (x#xs) = insert x (sort xs)"**

```
2. [| if a = x then True else x ∈ list; sort list ≠ [];
      x ∈ list ⟶ x ∈ sort list;
      if a = x then True else x ∈ list |]
   ⟹ x ∈ insert a (sort list)
```

- "insert x [] = []"
- **"insert x (h#t) = (if x ≤h then x#h#t else h#insert x t)"**
- *"sort [] = []"*
- "sort (x#xs) = insert x (sort xs)"

This identifies a program slice that has been involved in producing the False goal ([`insert_nil`, `sort_cons`]) and therefore assists in the hunt for errors.

In some similar proofs the step case is automatically discharged in which case $good(sort\_cons) = bad(sort\_cons) = 1$ and the rule's annotation becomes **"sort (x#xs) = insert x (sort xs)"** for the first goal leaving only `insert_nil` highlighted as "probably incorrect" giving further clues as to the culprit.

In this particular proof, attempts to prove the second goal lead to further proof branches that result in False conclusions attributable to `insert_nil` but also several branches that are discharged – overall $good(insert\_nil) = 0$ in all states while in general $bad(sort\_cons) = good(sort\_cons) + 1$. This suggests that the user may need access to further information about a rule's good and bad integers. Although it is unclear how such information can be conveyed by syntax highlighting alone, it would certainly be possible to introduce a further highlight for the "worst" rules (see §5) and/or to allow optional display of the good and bad values alongside the rules in which case `insert_nil` would be singled out in this example.

4.3 Getting Stuck

Assuming that `insert_nil` has been fixed, the last example we will consider picks up the main verification at a later stage. We will now assume that `insert` and `sort` have been removed from the suspect list. Two new functions and a new lemma have been introduced. `minl` returns the minimum element of a list of naturals and `-minl` returns a list with one occurrence of its minimum element removed. Among other things the following lemma has been established:

$$l \neq [] \Longrightarrow (sort\ l) = (minl\ l) \# sort(-minl\ l)$$

which when used in the proof leads to the goal

```
1. count_list x (Once (minl (a#list) #
                       sort (-minl (a#list)))) = 0
```

and the highlighted rules:

- *"Once [] = []"*
- "Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)
  else x1#x2#Once xs)"

A proof by cases follows on whether $-minl(a\#list)) = []$ simplification of the first goal leaves two subgoals of which the first:

```
1. [| a ≠ x & ¬ x ∈ list;
      (if a = minl (a # list) then list
                              else a # -minl list) = [];
       xs = a # list |]
   ⟹ count_list x (Once [minl (a # list)]) = 0
```

is associated with the following highlighted slices:

- *"Once [] = []"*
- "Once (x1#x2#xs) = (if x1=x2 then Once (x2#xs)
  else x1#x2#Once xs)"

While this doesn't directly highlight an error, the juxtaposition of the goal and the relevant rules, particularly with neither highlighted as used directly in the goal should prompt a user to recognise the omission of the relevant information.

5 Supportive Results

The ideas behind the interface design proposed here arise from work on the automated detection and repair of such errors within the proof planning framework [4]. Program slice tracking has been implemented in the $\lambda Clam$ [13] proof planning system. In the absence of an implementation in a theorem prover interface we report some results on the success of the heuristics within this system. We used a variation on the system reported in [4] [4] . That system attempts to repair erroneous rewrite rules. The system reported here simply terminated false branches and concluded the proof attempt by, for each rule, r, reporting $good(r)$ and $bad(r)$. Unfortunately some errors, especially those appearing in the recursive cases of definitions caused the system to be non-terminating, therefore an additional heuristic was used to close branches if the step case of an inductive proof could not be solved by appeal to the induction hypothesis [5] . We ran two experiments. In Experiment 1 closed step case branches did not contribute to the good/bad scores (ie. strictly adopting the conventions proposed in this paper). In Experiment 2 such closed branches increased the bad scores (arguably in a human proof attempt these branches would eventually have led to a False goal rather than the non-termination caused in $\lambda Clam$).

The table 2 shows the results for both sets of runs. The experiments involved 24 non-theorems based around errors in the definitions of list append, list membership and the insert and sort programs already covered in this paper. The theorems were

[4] Relevant code is available from the author on request.

[5] with the exception of a few special cases, for instance where the step case proof had branched following a case split

selected from the $\lambda Clam$ benchmark set rather than being actual specifications for these functions. As such these results should be considered indicative only. The tables report, for each experiment, whether the "incorrect" rewrite rule was underlined (ie. whether its good score was greater than its bad score) and the average number of rules underlined. This is the average number when at least one rule is underlined – in several cases no rule had a larger bad integer than a good integer. The intention in presenting this average is to provide evidence of the extent to which the heuristics help focus attention on an erroneous rule – after all it is not much help if *all* the rules are underlined. Including cases where no rule is underlined reduces this average and tends to suggest better discrimination than is actually the case. To follow this up we provide a percentage of the rules excluded. This is the percentage of the rules involved with definitions actually used in the proof which were not underlined. Again this only refers to situations where at least one rule was highlighted to give an impression of the extent to which the choices were narrowed down. False positives reports the number of situations where some rule was highlighted but the incorrect rule was not. We also computed an overall score for each rewrite rule as a tuple of the bad score and the good score. These tuples were then ordered according to $\succ$ where

$$(b_1, g_1) \succ (b_2, g_2) \iff (b_1 < b_2) \vee ((b_1 = b_2) \wedge (g_1 > g_2))$$

and $<$ and $>$ are the standard order on natural numbers. We report on the percentage of cases where the intended error was picked out by this heuristic and when it was the only rule with the highest score.

	Exp 1.	Exp 2.
Incorrect Rewrite Underlined	50%	66%
Average No. Rules underlined	1.62	2.11
Rules Excluded	52%	38%
False Positives	0	1
Incorrect Rewrite has Highest Score	62.5%	79.17%
Incorrect Rewrite has Unique Highest Score	29.17%	54.17%

Table 2
Summary of Experimental Results in $\lambda Clam$

The results show that it would be useful if the interface could also flag those rules which are scoring most highly under $\succ$ even where all rules are being used in more good branches than bad since this is clearly giving the best information about the location of errors.

The use of bad scoring for "stuck" goals (Experiment 2) is problematic – it improves the rate at which incorrect rules are identified, and the rate at which bad rules are highlighted as "worst" at the cost of losing discrimination (see Rules Excluded). Since the stuck heuristic is a crude attempt to mimic human "getting stuck" behaviour it is perhaps not surprising the effects are equivocal. At any rate

it is clear that, to a certain extent, this heuristic is too eager and prevents (in the first case) the proof from progressing to false branches that would (hopefully) later get scored if pursued by a human prover and, in the second case, generates too many false positives. Improving the heuristic is well outside the scope of this paper but interpretation of the above results need to bear its limitations in mind. The heuristic does suggest that there may be some benefit in allowing a human prover intervene in the scoring process and mark some branches as "bad" even where a False conclusion has not been reached.

Obviously these results are only indicative of how the heuristics might serve human users as opposed to an automated system but they do suggest that profitable use can be made of the information contained in program slices attached to proof branches. In particular the "worst" score looks particularly promising in terms of directing a user's attention to errors.

6 Related Work

The HAT tool [2] uses a mixture of algorithmic debugging and program slicing to direct a user's attention to relevant parts of a program's source. HAT creates an Evaluation Dependency Tree (EDT) tracing the execution sequence of function calls on a sample input. The nodes in this tree can be associated with their "call site" in the program. This allows the system to use a syntax highlighting mechanism to relate debugging traces back to specific parts of code. The tool works by identifying slices in the EDT and relating these back to the relevant portions of the code. This has recently been extended [3] to use a very similar polling system to that described above based on superimposing "correct" EDTs and "incorrect" EDTs to generate heuristic scores by which a "worst" slice can be identified.

In general the HAT tool only displays the most immediate redex rather than all those involved in a slice in order to reduce information overload – while it may be desirable to do something similar in proof-directed debugging it isn't at all obvious that the last rule to be used will generally prove to be the one at fault.

This is the first work I'm aware of that considers the use of proof tree branches as a slicing criterion or considers integrating the syntax highlighting interface of a debugging tool such as HAT into a Theorem Prover.

7 Further Work

7.1 Implementation

Clearly the most pressing and important piece of further work is providing an implementation of verification based program slicing to allow experimental evaluations of the extent to which it genuinely helps locate errors.

Out intention is to provide an implementation in Isabelle/Isar using the Proof General interface. This allows there to be a clean separation between the information used by the interface and that used by the underlying theorem prover. Such an approach also creates some challenges however, since the necessary properties of goals and proof states will have to be inferred. On the whole it should be relatively

straightforward to identify goals and key constants within goals although it there will be some challenges involved in keeping track of proof states, in particular the relationships between parent and child goals needed to make updates correctly. In Isabelle successfully discharged goals are dropped from the proof state presented to the interface which again is likely to raise some challenges in the tracking of information.

Although no examples have been shown here where a rule is used directly with a tactic (e.g. the `rule` method in Isar) this also needs to trigger updates of tracking information. In general this should be relatively straightforward based on simple analysis of tactic calls.

Simplification is the major step where the exact rules used by the system are effectively concealed from the user. It is also the most important tactic which can be used across multiple goals discharging some but not others (so leading to ambiguities about successful proof branches) and can generate and discharge new branches within its own application invisible to the user. Fortunately Isabelle's simplifier provides a tracing mechanism from which is it possible to infer rule usage and determine when a proof branch has been discharged, from which it should be possible to infer the necessary information. It may also be possible to use the proof object (of the top theorem) to track program slice information [6].

We have not considered how backtracking should interact with program slicing. At present the design assumes that proof states are generated in sequence and implicitly assumes that they can only be backtracked in that sequence. However many theorem provers allow backtracking on any open goal not just the those most recently derived. In this case it may be necessary for the interface to store additional information about the relationships between goals and their parents from proof state to proof state. This problem may also mean that ultimately it is cleaner to store program slice information in the prover's proof state rather than in the interface.

7.2 More Detailed Program Slices

So far we have considered program slices whose nodes are identifiable with the simple case structure of function definitions however there are further advantages to be gained if more sophisticated slicing is used in which function calls/sub-expressions are considered as nodes (as is common when applying program slicing to functional programs).

In the following example, again genuine, a student has been asked to provide a function, `removeAll`, which removes all occurrences of its first argument from its second. They appear to have programmed by analogy from a previous function, `removeOne`, where only one occurrence was to be removed and have forgotten to replace one call to this program. The code is expressed in Isabelle as:

```
primrec
   removeAll_nil: "removeAll x [] = []"
   removeAll_cons: "removeAll x (h#t) = (if x = h
                    then removeAll x t else h#removeOne x t)"
```

[6] My thanks to an anonymous referee for this suggestion.

Consider an attempt to establish that

$$\neg x \in removeAll\ x\ l$$

The proof proceeds by induction on `l` followed by simplification of all goals automatically discharging the base case and leaving the step case goal:

```
1. !!a l. ¬ x ∈ removeAll x l
          ⟹ (x = a → ¬ a ∈ removeAll a l) &
              (x ≠ a → a ≠ x & ¬ x ∈ removeOne x l)
```

and highlighted rules.

- *"removeAll x [] = []"*
- **"removeAll x (h#t) = (if x = h then removeAll x t**

  else h#removeOne x t)"

Use of some introduction rules (`impI` and `conjI`) and more simplification gives three subgoals which are based around a case split on whether `x = h` and then (following from a lemma about $\in$) on the values in the head and tail of `h#removeOne x t`. The first of these (where `x = h`) is automatically discharged leaving two subgoals, the first of which is

```
1. [| x ≠ a; ¬ x ∈ removeAll x l |] ⟹ a ≠ x
```

Ideally we would like to highlight the rules associated with this goal as follows:

- *"removeAll x [] = []"*
- **"removeAll x (h#t) = (if x = h** *then removeAll x t*

  else h `#removeOne x t)"`

showing that `removeAll x t` is probably correct and that this goal is based on the value of `h` in `h#removeOne x t`.

This goal is easily discharged leaving only the goal:

```
2. [| x ≠ a; ¬ x ∈ removeAll x l |] ⟹ ¬ x ∈ removeOne x l
```

Again ideally we would like to highlight parts of the second program slice differently:

- *"removeAll x [] = []"*
- **"removeAll x (h#t) = (if x = h** *then removeAll x t*

  else h# **removeOne x t)"**

Focusing attention on the problematic part of the rule which will eventually lead to False goals.

It should be easy enough to represent these slices within a system, for instance a simple list of integers can be used to indicate the position of a sub-expression within a rule and all sub-expressions of suspect rules stored in for use program slices. However it is much harder to see how information about which slice is relevant to a goal can be inferred without help by an interface such as Proof General. Indeed in order to supply the necessary information a theorem prover's internals may need modification in order to track the unifications performed when rules are applied in a meaningful way.

7.3 Imperative Programs

Obviously a long term objective is to extend this work to imperative programs. In these cases we lose the correspondence between program locations and rewrite rules. We would therefore need to adapt the concept of "used in a proof branch" to, for instance, identify individual program statements that had been involved in an instantiation of the assignment axiom in this branch of the proof.

8 Conclusion

This paper has discussed the use of verification as a program slicing tool. It has discussed how proof branches can be used to build up program slices based around equational rewrite rules and described a simple mechanism for deriving a heuristic score for how likely a given rule is to be correct. It has then discussed how such information might be presented to a user.

The mechanism proposed relies on a user identifying "suspect" rules. In the case study these all related to program function cases however there is no reason, in principle, why any definition or theorem in a theory could not be treated in the same way, allowing suspect specifications and definitions in general (non-verification based) proofs to be handled in the same way. The general mechanism can almost certainly be used in any situation where a reason is being sought for a proof failure.

Considerable further work, including an implementation, is required.

References

[1] D. Aspinall. Proof general: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000*, volume 1785 of *LNCS*. Springer, 2000.

[2] O. Chitil. Source-based trace exploration. In C. Grelck, F. Huch, G. J. Michaelson, and P. Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, LNCS 3474, pages 126–141. Springer, March 2005.

[3] T. Davie and O. Chitil. One right does make a wrong. In H. Nilsson and M. van Eekelen, editors, *Seventh Symposium on Trends in Functional Programming*, pages 27–40, 2006.

[4] L. A. Dennis. Program slicing and middle-out reasoning for error location and repair. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, editors, *IJCAR 2006 Workshop onDisproving - Non-Theorems, Non-Validity, Non-Provability*, 2006. To Appear.

[5] L. A. Dennis. The use of program slicing and middle-out reasoning to identify and repair program errors. Technical report, University of Nottingham, 2006. Submitted to IJCAR-2006 Workshop on Disproving - Non-Theorems, Non-Validity, Non-Provability.

[6] L. A. Dennis, R. Monroy, and P. Nogueira. Proof-directed debugging and repair. In H. Nilsson and M. van Eekelen, editors, *Seventh Symposium on Trends in Functional Programming*, pages 131–140, 2006.

[7] L. A. Dennis and P. Nogueira. What can be learned from failed proofs of non-theorems? In J. Hurd, E. Smith, and A. Darbari, editors, *TPHOLs 2005: Emerging Trends Proceedings*, pages 45–58, 2005. Technical Report PRG-RP-05-2, Oxford University Computer Laboratory.

[8] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In F. Baader, editor, *CADE19*, volume 2741 of *LNCS*, pages 279–283. Springer, 2003.

[9] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, 1992.

[10] R. Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):471–477, 1999.

[11] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[13] J. D. C. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with lambda-clam. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *LNCS*, pages 129–133. Springer, 1998.

[14] N. Rodrigues and L. Barbosa. Slicing functional programs by calculation. In *Proceedings of the Dagstuhl Seminar on Beyond Program Slicing*, 2005.

[15] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[16] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[17] M. D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[18] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99*, number 1690 in LNCS. Springer, 1999.