David Aspinall   Christoph Lüth (Eds.)

User Interfaces for Theorem Provers

International Workshop, UITP'03

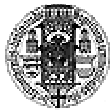
Rome, Italy

8th September 2003

Co-located with the

16th Conference for Theorem Proving for Higher Order Logics

TPHOLs 2003



Technical Report Nr. 189
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Foreword

This informal proceedings contains the papers and system descriptions presented at the *User Interfaces for Theorem Provers 2003* workshop (UITP 2003), held on Monday 8th September 2003, at the Università di Roma Tre, Rome, Italy, in conjunction with the TPHOLs conference.

The UITP workshop brings together researchers interested in designing, developing and evaluating interfaces for interactive proof systems. These systems include theorem provers, formal method tools, and other tools manipulating and presenting mathematical formulae.

The workshop series started in 1995 with a meeting in Glasgow (organized by Phil Gray, Tom Melham, Muffy Thomas and Stuart Aitken), and continued with meetings 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). The present organizers enjoyed these meetings and believe that the forum provides a unique opportunity to discuss work in this area, and so we organized this "revival" meeting for 2003. This met with strong interest in the community, as indicated by the range of papers in this volume.

The papers here show that during the five years in between, considerable technological advances have taken place, and progress has been made. Yet, the basic problems in this field remain far from solved. Moreover, with theorem proving and formal methods becoming more widely used, they become even more relevant, and make this a very exciting area to work in.

In keeping with tradition of previous meetings, the papers here have been briefly reviewed by a programme committee comprising

- Stuart Aitken,
- David Aspinall,
- Yves Bertot,
- Christoph Lüth,
- Tom Melham,
- Erica Melis and
- Burkhart Wolff.

We would like to express our gratitude to the other programme committee members for their help, and in particular to Burkhart Wolff for helping to arrange our meeting using the organizational machinery behind TPHOLs.

David Aspinall and Christoph Lüth.
Edinburgh and Bremen, July, 2003.

Table of Contents

Proof General meets IsaWin . . . 1
David Aspinall and Christoph Lüth

TeXmacs as Authoring Tool for Publication and Dissemination of Formal Developments . . . 14
Philippe Audebaud and Laurence Rideau

Visualizing Geometrical Statements with GeoView . . . 32
Yves Bertot, Frédérique Guilhot and Loïc Pottier

Rendering Tree Proofs in Box-and-Line Form . . . 46
Richard Bornat

Interactive disproof . . . 62
Richard Bornat

Taclets and the KeY Prover . . . 74
Martin Giese

Interactive Proof Construction at the Task Level . . . 81
Malte Hübner, Christoph Benzmüller, Serge Autexier and Andreas Meier

Improving the PVS User Interface . . . 101
Joseph R. Kiniry and Sam Owre

Proving as Programming with DrHOL: A Preliminary Design . . . 123
Scott Owens and Konrad Slind

User Interface for Adaptive Suggestions for Interactive Proof . . . 133
Martin Pollet, Erica Melis and Andreas Meier

Formal Proof Authoring: an Experiment . . . 143
Laurent Théry

Thoughts on Requirements and Design of User Interfaces for Proof Assistants . . . 160
Norbert Völker

Author Index . . . 176

Proof General meets IsaWin

David Aspinall[1] and Christoph Lüth[2]

[1] LFCS, School of Informatics, University of Edinburgh, U.K.
WWW: `http://homepages.inf.ed.ac.uk/da`
[2] Department of Mathematics and Computer Science, Universität Bremen
WWW: `http://www.informatik.uni-bremen.de/~cxl`

Abstract. We describe the design and plan for implementing a combination of theorem prover interface technologies. On one side, we take from Proof General the idea of a prover-independent interaction language and its proposed implementation within the PG Kit middleware architecture. On the other side, we take from IsaWin a sophisticated desktop metaphor using direct manipulation for developing proofs. We believe that the resulting system will provide a powerful, robust and generic environment for developing proofs within interactive proof assistants, and we intend to demonstrate this for some of our favourite systems.

1 Introduction

Proof General is a generic interface for interactive proof assistants, built on the Emacs text editor [3, 2]. It has proved rather successful in recent years, and is popular with users of several theorem proving systems. Its success is due to its genericity, allowing particularly easy adaption to a variety of provers (primarily, Isabelle, Coq, and LEGO), and its design strategy, which targets experts as well as novice users. Its central feature is an advanced version of script management, closely integrated with the file-handling of the proof assistant. This provides a good work model for dealing with large-scale proof developments, by treating them similarly to large-scale programming developments. Proof General also provides support for high-level operations such as proof-by-pointing, although these are less emphasised.

Proof General has some drawbacks, however. From the developers' point of view, it is rather too closely tied with the Emacs Lisp API which is somewhat unreliable, often changing, and exists in different versions across different flavours of Emacs. From the users' point of view, although the interface offers some high-level operations and tries to hide the shell-window process, interaction is still firmly based on the proof assistant's (often cryptic) command language, and not much support is offered for specific tactics or commands.

IsaWin is the instantiation of a generic graphical user interface to Isabelle [16, 15, 17]. It aims at providing an abstract user interface based on a persistent visualisation metaphor and the concept of direct manipulation; users need not be concerned with the syntactic idiosyncrasies of the prover, and can instead concentrate on the logical content of the proof. This abstract approach also

allows high-level operations: for example, there is support for proof-by-pointing (folding or unfolding equations), term annotations (the system can display the type of sub-terms), or tactical programming (one can cut parts from the history of a proof to make it into an elementary tactic, which can be reapplied elsewhere).

While IsaWin is usually met with initial approval, in particular from novice users, it has some drawbacks. Customising or adapting it to other proof assistants requires Standard ML programming, and a good understanding of the data structures, so it is not possible to adapt a system gradually (unlike with Proof General), and it is hard to adapt to provers not implemented in Standard ML. Moreover, it only has a rudimentary proof script management, and integrates foreign proof scripts only reluctantly.

Thus, Proof General and IsaWin can be considered as complimentary, with each offering advantages to compensate for the other's shortcomings. This paper describes an attempt to combine their advantages in one system based on the PG Kit infrastructure presented in [4, 5]. The resulting system has an event-based architecture and focuses on managing proof scripts as the central artefact. A generic interactive protocol language allows one to connect different user interfaces (called *displays*); thus, IsaWin becomes one particular user interface in this setting.

In the remainder of this paper, we describe the particular aspects of Proof General and IsaWin that we want to build on, followed by a outline design of the new system. We close by briefly mentioning related work, and our vision for the future evolution of the project.

2 Proof General Basic Concepts

Emacs Proof General is the present incarnation of Proof General, based on the ubiquitous Emacs text editor.[3] For Proof General, a proof consists of the user-editable formal text (or *proof script*) which, when processed by the machine, constructs a representation of a proof in a formal system. The target proof script is the central focus of development.

It doesn't matter whether the proof script contains a tactic-style procedural proof, or a declarative proof description. The interface relies on the underlying proof assistant being able to process a proof script interactively and incrementally, in a textual dialogue: the user issues a proof command, and the system responds. A distinction is made between *proper* proof script commands, which belong in the text, and *improper* commands, which do not. Proper commands include statements of propositions to be proved, and the contents of their proofs. Improper commands include undo steps, commands for inspecting terms and for querying a database of available theorems. The proper commands are stored in the proof script file, and coloured according to progress in the proof: crucially, regions which have been processed by the proof assistant are coloured blue and should not be edited. This is the idea of *script management* as described in [8].

[3] Several flavours of Emacs are supported, including XEmacs and GNU Emacs, running on numerous platforms.

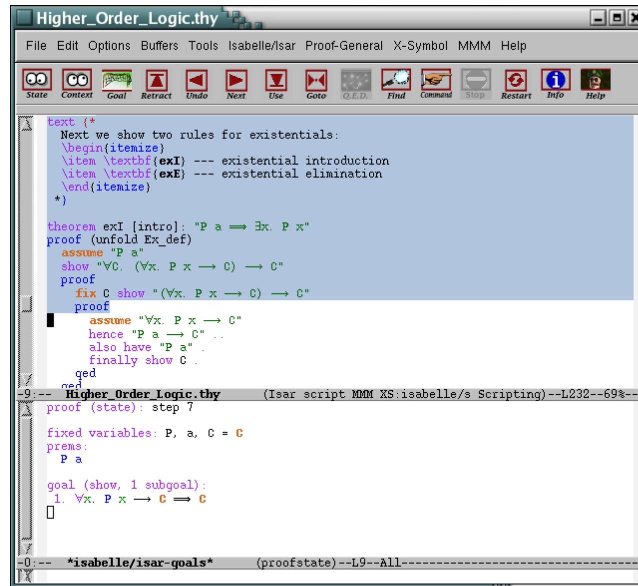


Fig. 1. The Proof General interface. The window is split into two panes, the bottom displaying the output from the proof assistant, the top displaying the proof script, coloured according to progress.

Proof General provides a simple browsing metaphor for replaying proofs, via a toolbar for navigating in a proof script, see Fig. 1 for a screenshot. Behind the scenes, this works by sending commands to issue proof steps and undo proof steps. The undo behaviour relies on the built-in history of the proof assistant, which typically forgets the history between steps of a proof once the proof has been completed.[4] The undo behaviour splits the proof script into regions (or *spans*) of consecutive lines which are atomic for undo. Spans are treated linearly within the file, although they have a dependency structure which expresses dependencies within the proof development itself: it is possible to display this dependency within the interface, to highlight the auxiliary lemmas used in proving a theorem, for example.

Proof General provides an advanced implementation of script management which synchronises file editing between the proof assistant and editor in a two-way communication. Files are used to store proof scripts, representing parts of developments. If the user completes processing a proof script file, Proof General informs the prover; if the prover reads another file during processing, it will inform Proof General. Similarly for undo operations at the file level.

Technically, Proof General is implemented mostly in Emacs Lisp, interfacing with other Emacs packages, notably including X-Symbol [26] for displaying

[4] This is initially counter-intuitive to new users, although they soon get a feel for it.

mathematical symbols. A considerable effort has been made into making it easy to adapt Proof General to new proof assistants, and it can be possible to configure by setting only a handful of variables, with little or no Emacs lisp programming. But the mechanism is tiresome: we try to anticipate and cater for many different proof assistant behaviours within Proof General. Moreover, for advanced features (such as proof-by-pointing [7] and proof-dependency visualisation [18, 21]), dedicated support from the proof assistant is inevitably required.

3 Proof General Kit Basic Concepts

To address the limits of the Proof General model, and particularly of the Emacs Lisp implementation, Proof General (PG) Kit has been designed [4, 5]. The central idea is to use the experience of connecting to diverse provers to prescribe a uniform protocol for interaction. Instead of tediously adapting Proof General to each prover, Proof General will call the shots, by introducing a uniform *protocol for interactive proof*, dubbed **PGIP**, which each prover must support.

As well as controlling the progress of the interactive proof, PGIP is responsible for initialization and book-keeping tasks which require communication between the proof assistant and the interface architecture. It includes messages to configure (proof assistant specific) user-level menus and preferences; messages to display status or error dialogues, and messages to retrieve theory files. PGIP is based on messages sent in an XML format.

Apart from communicating the intent of a proof interaction, we want to communicate a representation of it; for example, by displaying a list of subgoals to be solved to complete some proof. To help with this, PG Kit includes another XML format, **PGML**, the Proof General Markup Language. The markup language is intended for displaying *concrete* syntax. It includes representations for mathematical symbols, along with the possibility of hidden annotations which express term structure. These annotations can be used to implement sub-term cut-and-paste, proof-by-pointing or similar features [7]. Of course, there are already languages for XML-based document formats designed for displaying mathematics (MathML, [24]), and transferring mathematical content between applications (OpenMath, [20]). Later on, we hope to use these languages with PGML. At the moment, it is more feasible to implement our own simple markup scheme since both MathML and OpenMath go further into the *abstract* structure of terms than we need, or than we can easily accommodate in a generic way.

PGIP and PGML are described in full elsewhere [5], including DTDs for the languages, and the description of the architecture into which they fit (described more in Sec. 5). There is not yet a complete implementation of Proof General Kit, but experimental progress on several fronts.[5] The work described below will be a major advance.

[5] Isabelle2003 uses PGIP messages to configure PG's menus, and supports PGML output.

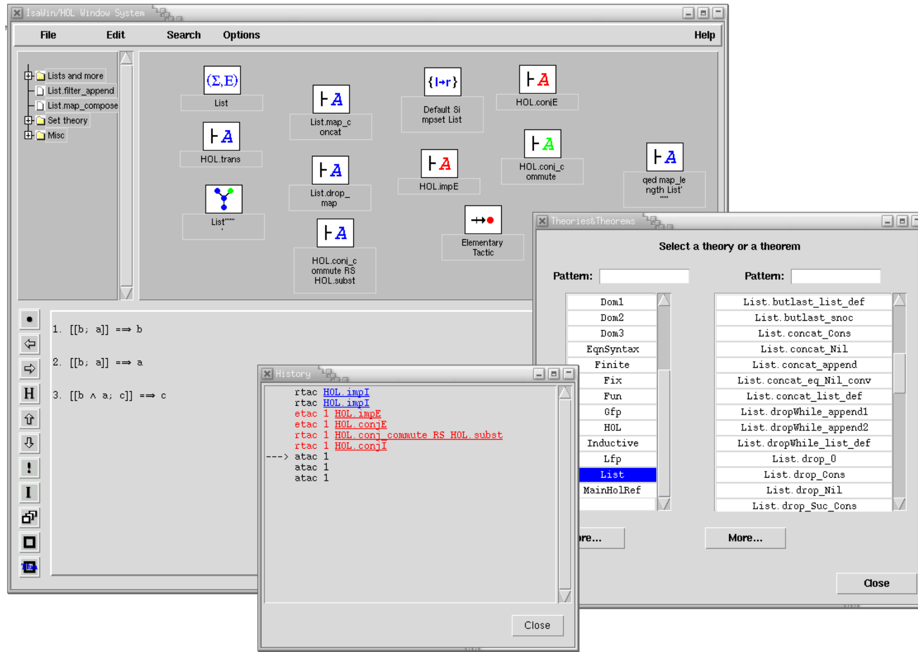


Fig. 2. The IsaWin interface. The main window shows the notepad and folder navigation bar on the top, and the ongoing proof below. Two auxiliary windows open here allow the search for theorems, and show the history of the proof.

4 IsaWin Basic Concepts

IsaWin is the instantiation of a generic graphical user interface called GenGUI for the Isabelle proof assistant. It provides a more abstract, less syntax-oriented interface to Isabelle (and related provers), based on the visual metaphor of a *notepad*. All objects of interest, such as proofs, theorems, tactics, sets of rewriting rules etc. are visualised by icons on a notepad, and manipulated there using mouse gestures. More complex objects such as proofs can be manipulated by opening them in a separate sub-window. IsaWin offers self-contained history support, proof-by-pointing, dependency management and session management.

The interface is based on the concept of *direct manipulation*: a continuous representation of the objects and actions of interest with a meaningful visual metaphor and incremental, syntax-free operations, i.e. user gestures such as dropping an object onto another, pop-up menus or mouse clicks. Objects are visualised by icons on a *notepad* (see Fig. 2). The icon is given by the type of the object, which determines the available operations. Hence, when users see an object visualised by a theorem icon, they know that if they drop this object onto another theorem object, the theorem prover will attemp to combine them by forward resolution, whereas if the theorem is dropped onto on ongoing proof,

a new proof step using backward resolution with this theorem will be attempted. The type of an object further determines the operations appearing in its pop-up menu and whether (and how) it can be opened into a separate sub-window. The interface also keeps track of dependencies, i.e. if one object is used as an argument to construct another object, and if objects are changed or outdated, all dependent objects are outdated as well.

Drag&drop is resolved by a table indexed with the types of the dropped and the receiving object respectively. The history is represented internally, as the sequence of operations used to construct an object. This has some advantages, as it allows an abstract manipulation of proofs; for example, we can cut out parts from proof scripts and make them into reusable tactics. But moreover it has severe disadvantages: as proof scripts are an external representation of the history, a generic script management is not very straightforward to implement, and indeed IsaWin only offers limited script management, compared to Proof General. IsaWin has its own format to save and read proofs, and while it has some limited support to produce proof scripts in Isabelle's native format, it cannot parse them. This makes it hard to use IsaWin on proof scripts not developed using IsaWin.

Technically, GenGUI is an implemented completely in Standard ML (SML). IsaWin is the instantiation of GenGUI with a particular Isabelle-specific module. To customise it (adding or changing icons, shortcut buttons, or menu entries), one has to change this module. To adapt GenGUI to another proof assistant, one implements a different module with a given signature, containing the object types, operations and the drag&drop table. This requires a good understanding of the signature (and of SML), and also makes adaption to provers not implemented in SML cumbersome.

For us, the experience with GenGUI and IsaWin has shown that the implementation of an interface as an add-on in the same programming language as the proof assistant leads to a non-modular architecture, which makes the interface difficult to reuse. It also makes the interface less robust: if the prover diverges or returns a run-time error, the interface diverges or stops as well. For these reasons, not many different adaptations of GenGUI exist, and in comparison with Proof General, GenGUI has not fully delivered on its promise of genericity.

A better architecture is to keep interface and proof assistant separate, and specify their interaction cleanly and in a language independent way — which is precisely what PGIP does. Proof scripts, understood broadly as the sequence of proof steps leading to a goal, are the main artefacts the user is constructing when working with a proof assistant, and as such should be represented and manipulated explicitly, rather than implicitly as in GenGUI.

5 System Architecture

The system is a synthesis of two existing designs. From the users' point of view, nothing much should change; their experience when working with the new system should by and large by the same as working with IsaWin or Proof General.

Hence, the main novelty of this system will be its internal architecture and communication protocols, which we will be describe in this section.

The system is an implementation of the PG Kit architecture presented in [4], which comprises a central *proof mediator*, connected to several *displays engines* and a *proof assistant*. The different components communicate in the interface protocol language PGIP (see above).

The mediator maintains the overall state, the proof scripts, and their dependencies. It sends commands to, and receives status messages from the proof assistant; at the same time, it receives user input from the display, and keeps the displays up-to-date. In principle, we can connect to more than one proof assistant at the same time, but presently we cannot interchange any data (such as proofs or theorems).

A display engine or display for short visualises the proof assistant's state, proofs, theorems, and so on. There can be more than one display connected to the main broker, and there are different kinds of displays. A simple *line-oriented display* displays lines of text and relays command lines typed by the user; this corresponds to Emacs Proof General. Slightly more sophisticated, a *graphical display* translates user gestures (drag&drop and so on) into textual prover commands; thus, IsaWin becomes another specific display module in this architecture. Other possible displays could include a web interface (either client-side by running an applet in a browser, or server-side by embedding the mediator into a web server via techniques such as servlets, ASP or CGI). We further distinguish between *active* and *passive* displays. An active display allows the user to enter commands, whereas a passive display primarily visualises proofs, possibly allowing active browsing.[6]

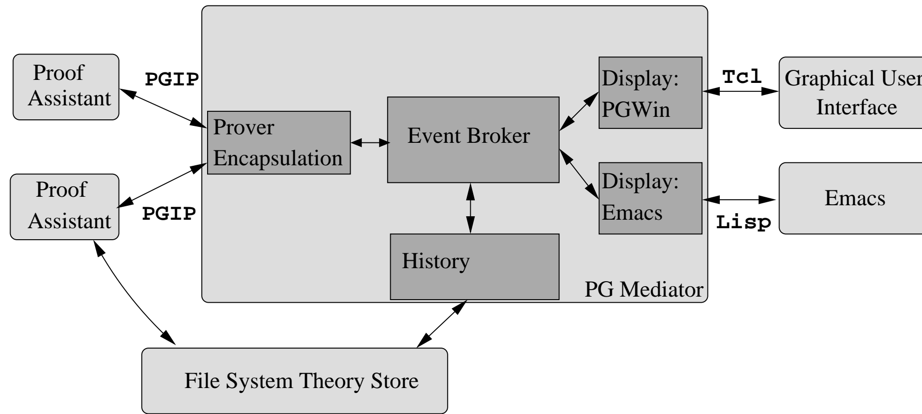


Fig. 3. System Architecture

[6] In [4], these have been called input engines and display engines.

Fig. 3 shows the system architecture. Rounded boxes denote separate components, and square boxes denote the different modules of the mediator. The implementation is based on the notion of an *event*. Events are generated either from user input, or change of state in the prover. The mediator is the central event broker; it receives input events, updates its internal state, and sends on change events to other parts of the system as necessary. Events are structured: they contain PGIP messages.

5.1 Implementing the Mediator

The mediator is implemented in Concurrent Haskell, using the Glasgow Haskell Compiler, with events as first-class values in Haskell [22]. That is, we have a polymorphic datatype of events containing a value of any type, with operations to synchronise on an event, sequence an action with an event, combine two events via deterministic choice, and others:

```
Event a
sync :: Event a -> IO a
(>>>=) :: Event a-> (a-> IO b)-> Event b
(+>)   :: Event a-> Event a-> Event a
```

This allows us to write concurrent functional programs in a notation reminiscent of process algebras such as the π-calculus [19].

We implement the mediator as a set of Haskell modules, loosely coupled (by Unix pipes) to the other components. The mediator contains one central event broker, and one event handling module for each loosely coupled module, such as displays and proof assistants (see Fig. 3). This architecture combines the advantages of loose and tight coupling, and offers the flexibility of a component-based framework such as CORBA without its implementation and performance overhead.

Events contain PGIP messages. To model PGIP faithfully in Haskell, we use HaXML [25]. From a given DTD, HaXML generates a series of Haskell datatypes, one for each element, along with functions to read and write XML. The advantage is that the type security given by the DTD extends into our program, making it nearly impossible to send messages containing invalid XML, and detecting the reception of invalid XML immediately.

There are different types of events, corresponding to different elements in the PGIP DTD, represented by different types in the mediator. These include:

- *Command events* (type `DispCmd`) are commands input by the user. Commands are generated by active displays, either by interpreting gestures or by the user typing a command line (or in fact mixtures of the two), and then handled by the event broker.
- *Display message events* (`DispMsg`) contain messages to be displayed, such as a new proof assistant state, error or warning messages. Display events are generated by the event broker from proof response events, or as an answer to user input (e.g. trying to browse beyond the end of the history).

– *Prover commands* (`ProverCmd`) are commands sent to the prover encapsulation, either generated from command events, or when replaying proof scripts. They contain PGIP which is sent on the corresponding proof assistant.
– *Prover message events* (`ProverMsg`) are messages from the proof assistant in response to proof commands, containing e.g. a new proof assistant state, or an error returned by the prover. They are interpreted by the event broker, which updates its internal states accordingly, and updates the connected displays.
– *Display configuration events* (`DispConfig`) configure the display, adding or changing menus, shortcut buttons, icons, or the drag&drop table. These are sent by the proof assistant to the broker, and passed on to all connected displays.
– An *interrupt event* (`DispINT`) signals that the user wants to interrupt whatever the proof assistant is doing currently. Interrupt events are ordinary events to the event broker, but they cause the module handling the proof assistant to send an out-of-band message, to the proof assistant, such as a user interrupt signal (SIGINT) on Posix systems.

To illustrate the event concept, we follow a typical event around the system. Assume that the user enters a command, e.g. by pressing a button on the GUI or merrily typing on his keyboard. This command results in a `DispCmd` event being sent from the display to the event broker. The event broker looks up the current proof from its internal state, and which prover handles this proof. It sends a `ProverCmd` event to the prover encapsulation, updates the display with the new command, and appends the command to the history. Some time later, the proof assistant returns a result, generating a `ProverMsg` event which is sent to the event broker. If it is an error message, all outstanding commands to that prover are flushed (if one proof command fails, it usually makes no sense to try subsequent ones). If the command succeeds, the prover's new state will be noted. In any case, a `DispMsg` event containing either the error message, or the new prover state, will be sent to all displays.

As we can see, this is asynchronous: users can type ahead of the prover. This way, proof replay is not any more different from normal user input, and both can be handled in a uniform way.

5.2 Displays

All display engines, even such seemingly different ones like Emacs and IsaWin, serve to visualise display messages originating mainly from the event broker. Crucially, later display messages refer to earlier ones. For example, in a line-oriented interface, when we replay a proof, we want to mark previously display lines as "replayed" (by colouring them differently). Moreover, user input pertains to the previously displayed messages: in a line-oriented interface, users may change a line in a proof, and in a graphical interface, users may click on icons, or drag and drop them.

In order to subsume different display engines in a uniform framework, we use the notion of an *object* as introduced by the generic graphical user interface

(see Sect. 4 above). An object is anything the system needs to keep track of: most prominently, ongoing proofs, but also auxiliary objects such as theorems, tactics, rewrite rules, and so on. In PGWin, an object corresponds to an icon on the desktop; in Emacs, an object corresponds to a *span*, which is a particular line range in a particular text buffer.

All display engines have to implement certain operations, such as display warnings, errors and status messages; create a new object; update a previously displayed object; outdate an object; receive user input; etc. Technically, we define a type class `Display`, which defines the operations that a display has to implement, for example creating and updating objects, generating command events (`DispCmd`), and receiving display message events (`DispMsg`):

```
class Display d where
  newObject :: d-> ObjId -> ObjType -> DispAttr -> IO ()
  updObject :: d-> ObjId -> ObjText -> IO ()
  bind      :: d-> IO (Event DispCmd)
  send      :: d-> DispMsg-> IO ()
```

Every kind of display engine is modelled by a different type, but all are instances of the display class:

```
instance Display PGWin where
  -- definition of functions follows ...
instance Display Emacs where
  -- definition of functions follows ...
```

All displays are kept in one heterogeneous list (using existential types [14]): display events are sent to all elements of this list, and the command events generated by all displays are the command events of each display combined with the obvious extension of the binary choice operator `(+>)` to lists of events.

5.3 PGWin

As explained above, IsaWin is organised around types and objects. User gestures are translated into commands by a table indexed with the types of the objects. In IsaWin, this table was implemented as an ML structure. Here, this table, along with the types of the proof assistant, the icons and many other details of the visual appearance are determined by display configuration messages, which form part of PGIP. Thus, when connecting a proof assistant to the mediator, the proof assistant sends configuration messages which tell the system the types of objects the proof knows about, and the commands triggered by drag&drop. There are further display configuration messages to add, change, or delete a custom menu's entries, configuration options, or shortcut buttons. This flexibility can also exploited by the user, so the user can change the appearance of the desktop on-the-fly by adding or deleting shortcut buttons or menu entries with user-defined tactics attached to them.

The display handler translates user gestures into prover commands, which are textually represented, stored and displayed. As opposed to IsaWin, where

commands were stored in an internal abstract representation, this has distinct advantages: it accustomises users to the syntax of the command language, since they will see it appearing in the history, it allows users to textually enter commands as well, giving them full access to proof tactics or customised proof procedures, and finally, it allows mixed textual and graphical user input.

The PGWin display engine is implemented in HTk, a functional encapsulation of the graphical user interface library and toolkit Tcl/Tk into Haskell [12]. Thus, the PGIP events are translated into Tcl code within Haskell, which is then sent on to the Tcl/Tk interpreter wish (see Fig. 3). A future, alternative graphical display engine might communicate externally in PGIP; but there is still a considerable amount of implementation work to organise the graphical interface which we believe is better done in Haskell than in an untyped scripting language like Tcl.

6 Conclusions and Outlook

This paper has described the concepts underlying the synthesis of the Proof General and IsaWin interfaces into one combined interface. The new interface has an event-based architecture based on the PG Kit, and consists of several components communicating in the PGIP proof protocol. In the first instance, we view this advance largely as a refactoring of existing designs: the user experience when working with the new system be broadly similar to the present IsaWin. However, the aspects that will make the system more useful to expert users will be improved as mentioned above, for example, in the ability to view and import proof scripts. Moreover, the open architecture opens the way to more easily implementing new interaction mechanisms, and developing a truly generic high-level interface.

State of the implementation. Currently, implementation of a first prototype to be demonstrated at UITP'03 is under way; thus, it is too early to say anything about the usability and success of the resulting system. However, we believe that the concepts introduced and explained in this paper are of sufficient general interest.

Related Work. Many other projects such as OMEGA [6], OMRS [10], Prosper [9], and HELM [1] also address the problems of interoperability. OMEGA has developed into the MathWeb project, providing an web-based mathematics infrastructure, which is perhaps closest in scope to our work. MathWeb uses an an XML-based interchange format called OMDoc [13] to communicate mathematical documents (proofs, theories, and so on). While OMDoc is targeted towards exchanging documents, and not interactive proof, enabling our system to exchange OMDoc documents, and thus connect it to MathWeb would be beneficial: it would allow MathWeb to make use of interactive proof assistants, and would allow PG Kit to tap into the vast supply of MathWeb documents. Prosper uses another approach to interoperability, aiming at connecting different automated proof tools together: at the core of the system an LCF prover kernel is

used to guarantee logical consistency. This is a more logic-centred view, with an emphasis on the exchange of proofs and theorems, and hence complimentary to our work.

Outlook. At the moment, proof scripts are still stored in the proof assistant's native format. A future item on the agenda is the definition and implementation of a generic scripting language, which would allow the exchange of *replayable* proofs, or even parts of proofs, between different proof assistants. This would also allow us to consider generic manipulation of proof developments, such as dependency analysis or lemma-lifting. Adding foreign exchange formats such as OMDoc would be a worthwhile addition as well, as it extends the scope of our system. More technically, the use of a versioning repository to store proofs would be a useful feature for advanced users; hopefully, a free off-the-self solution such as Subversion [23] could be used here.

Acknowledgements. DA was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

References

1. Andrea Asperti et al. HELM: Hypertextual electronic library of mathematics, 2002. See web page hosted at `http://helm.cs.unibo.it/`, University of Bologna.
2. D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General, 2003. System documentation, see `http://www.proofgeneral.org/doc`.
3. David Aspinall. Proof General: A generic tool for proof development. In Graf and Schwartzbach [11], pages 38–42.
4. David Aspinall. Protocols for interactive e-proof, 2000. Available from `http://www.proofgeneral.org/doc`.
5. David Aspinall. Proof General Kit. White paper. Available from `http://www.proofgeneral.org/kit`, 2002.
6. Christoph Benzmüller et al. ΩMega: Towards a mathematical assistant. In William McCune, editor, *14th International Conference on Automated Deduction — CADE-14*, LNAI 1249. Springer, 1997.
7. Yves Bertot, Thomas Kleymann, and Dilip Sequeira. Implementing proof by pointing without a structure editor. Technical Report ECS-LFCS-97-368, University of Edinburgh, 1997. Also published as Rapport de recherche de l'INRIA Sophia Antipolis RR-3286.
8. Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7):161–194, February 1998.
9. Louise A. Dennis et al. The PROSPER toolkit. In Graf and Schwartzbach [11].
10. Fausto Giunchiglia, Paolo Pecchiari, and Carolyn Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems. In F. Baader and K.U. Schulz, editors, *"Frontiers of Combining Systems - First International Workshop" (FroCoS'96)*, Kluwer's Applied Logic Series (APLS), pages 157–174, 1996.

11. Susanne Graf and Michael Schwartzbach, editors. *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2000*, LNCS 1785. Springer, 2000.
12. HTk — graphical user interfaces for Haskell programs, 2002. `http://www.informatik.uni-bremen.de/htk`.
13. Michael Kohlhase. OMdoc: Towards an OpenMath representation of mathematical documents. Available from `http://www.mathweb.org/omdoc/`.
14. Konstantin Laufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485 – 517, May 1996.
15. Christoph Lüth, Haykal Tej, Kolyang, and Bernd Krieg-Brückner. TAS and IsaWin: Tools for transformational program developkment and theorem proving. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering FASE'99*, LNCS 1577, pages 239– 243. Springer, 1999.
16. Christoph Lüth and Burkhart Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167– 189, March 1999.
17. Christoph Lüth and Burkhart Wolff. More about TAS and IsaWin: Tools for formal program development. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering FASE 2000*, LNCS 1783, pages 367– 370. Springer, 2000.
18. Fiona McNeil. On the use of dependency tracking in theorem proving. Master's thesis, Division of Informatics, University of Edinburgh, 2000.
19. Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
20. The OpenMath Society. `http://www.nag.co.uk/projects/openmath/omsoc/`.
21. Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *Proc. User Interfaces for Theorem Provers, UITP'98*, 1998.
22. George Russell. Events in haskell and how to implement them. In *International Conference on Functional Programming ICFP'01*, 2001.
23. Subversion. A compelling replacement for CVS. Available from `http://subversion.tigris.org/`, 2003.
24. Mathematical markup language (MathML). W3C Recommendation, 1999. `http://www.w3.org/TR/REC-MathML/`.
25. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming ICFP'99*, pages 148– 159. ACM Press, 1999.
26. Christoph Wedler. Emacs package X-Symbol. Available from `http://x-symbol.sourceforge.net`, 2003.

TeXmacs as Authoring Tool for Publication and Dissemination of Formal Developments

Philippe Audebaud and Laurence Rideau

Lemme Project – INRIA
2004 route des lucioles - BP 93
FR-06902 Sophia Antipolis
email: {paudebau,lrg}@sophia.inria.fr

Abstract. We present an authoring tool for publication and dissemination of formal theories certified by a proof assistant. We have used TeXmacs, a "wysiwyg" editor for scientific documents which allows a connection with symbolic systems. We evaluate this tool on our formal developments certified using Coq.

1 Introduction

We present a toolkit named TmCoq helpful for users who have validated some formal developments with a theorem prover assistant, such as Coq [7] and who want to ready their work for a scientific publication or online browsing. As such, we assume the user's main requirements are, firstly, to be able to present their work in the usual language for a mathematical article and, secondly, to preserve the property that the underlying development still remains certified by the prover. For the first goal, we offer a powerful tool which provides LaTeX quality typesetting for presenting the formal development and its documentation together. To preserve correctness of the development at each step, our approach to the second goal is to work always on the whole development, but to provide different *view*s on this development. For documentation purposes, the user would prefer to be able to see the whole development, while the article is a selected view on the fully documented development, displaying only fragments of the whole development code.

A first solution that comes to mind consists in writing a LaTeX document. This is how one typically diffuses his results, inserting in places selected pieces of the formal development. This method has obvious drawbacks. The cut and paste process may introduce errors when the formal development had none. Any modification in the development requires the synchronization in the article, otherwise making the latter inaccurate or even wrong with respect to the certification step performed using the theorem prover. The inserted pieces of source code are introduced verbatim, which means that the syntax could be quite far from usual mathematical language and natural language as well. Notation and informal explanations are thus very different from the formal text. The theorem prover syntax may even change, introducing another source of misunderstanding.

To find a better solution to our problem, let us have a closer look at what makes up such a formal development in Coq. It consists of one or several files, called *scripts*, whose structure follows the usual one from a programming point of view. Declarations, statements, commands specific to the prover, etc. are part of the *vernacular sentences*. They form the *active part* of the script which means they are the only parts of the script that the theorem prover needs to know anything about. Any piece of documentation or explanation placed throughout the code can only be placed in the *comments* fields, which are irrelevant (i.e., *inactive*) from the prover's point of view.

Note that many documentation tools have been developed thus far for programming languages. Javadoc and Doxygen (among others) have made interesting contributions for this purpose. However, they are much too code-centric to be useful in our context. The users want to use LaTeX for writing, documenting, and publishing their work since a formal development consists of definitions, theorems, and proofs rather than pieces of programs. Nevertheless, a starting point is simply putting TeX commands within comments as part of the scripts.

On the Theorem Prover Assistant side, a LaTeX generation facility is offered by Pvs [5] for displaying formulas and proofs. It is based on an easy to use *substitution mechanism*. MetaPRL [6] provides (low level) formatting instructions based on the formatting library of its OCaml implementation language, together with PostScript and HTML output formats, but no LaTeX code generation as far as we know. In Coq context, this functionality is performed by Coqdoc [4], which offers LaTeX and HTML as output formats from scripts. This tool makes a simple lexical analysis which is sufficient for the generation of hyper-links for identifiers and index tables for the various sorts of statements. The vernacular sentences are simply inserted *verbatim* in the generated document. Such a tool still fails to present the formal development fragments in a user-friendly syntax, as independent as possible from changes on the Coq side. Also, documentation and code still belong to different worlds. It is not possible to make forward references, insure uniform notation throughout the generated document. While the process has been automated, thus less prone to human errors, the result is still bound to the version of Coq used for the certification.

1.1 Objectives

The dissemination of formal developments (in scientific articles or by code distribution on the Web) is an important part of the development's life, so we want to make a tool to help the user perform this task with the assurance that the correctness of the presented source code is preserved.

This tool must help the user write code documentation (e.g., writing comments in programs) and it also must help the user to produce scientific articles by allowing code importation and by making it easy to add natural language explanations. We want to offer various output formats, for example, allowing the consulting and browsing of the commented code through a Web interface (i.e., the tool has to manage navigation links in the code) and offering all stan-

dard output formats used in scientific document publishing: LaTeX, HTML, PostScript, etc.

The tool must provide high-quality visualization, using notation that is familiar to the reader (e.g., using standard mathematical notation) as well as a user-friendly interface. It also must allow one to present the formal developments using a natural syntax and must avoid whenever possible the use of a given theorem prover's scripting language. That is, the interface should display $\forall a : \mathrm{nat}, \sqrt{a^2} = |a|$ rather than `(a:nat)(equal (sqrt (power a (2))) (absolu a))`. The user must be able to define his own notation, and this notation must be kept homogeneous throughout the various parts of the document (i.e., keep the same notation in the explanations as in the imported code fragments).

Finally, to avoid a common source of errors in the translation from the source code to the presentation (typos are always unpleasant but are particularly annoying in the context of certified developments), the tool must allow the *automatic* importation of the source code (for example the theorems statements, the definitions, etc.) to preserve these statements' correctness. This automatic translation should also make it possible to preserve links, connections, correspondences back and forth between the presentation and the scripts produced, and controlled, by the prover.

Automatic translation preserves the visualized code's correctness as long as the user does not modify it. Here, one can import only fragments of the certified code in the final document. But, as soon as, one needs to modify the imported code, to assure the correctness means that our tool has to manage the whole development (even if only fragments are displayed) either to allow an extraction of the development for certification outside of the tool or to allow an interactive certification inside the tool, communicating directly with the prover. Actually, unlike a fragment of programming code that be considered independently of the rest of the code, a fragment of formal development can only be certified in the context of the whole development. Figure 1 shows the documentation process with the initial importation of the script, followed by the documentation steps, and finally the extraction into various formats in particular the Coq script extraction, allowing another cycle of certification.

The Coq script target is important. First, for compatibility reasons, we must provide a stand-alone version of the script, where the documentation fragments are inserted in the appropriate comment regions. Second, and more importantly, this target is evidence that one can extract from the enriched document the parts which concern the user formal development, this new script must be accepted by the prover afterwards.

1.2 Some examples

Our tool is not tightly coupled to a given theorem prover, in the future it should even be an independent tool, however, we begin with a case-study of our formal developments which are mainly done using the Coq theorem prover. Thus, the presented examples use Coq syntax.

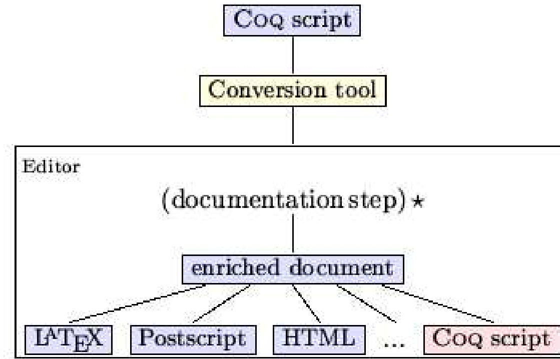


Fig. 1. Generation of the documentation from a Coq script.

Let us take a simple example. Note that in Coq concrete syntax `(X:A)B` stands for the the universal quantification $\forall X : A.B$, `~A` for the logical negation $\neg A$, while `(EXT x:C | P)` is meant for the existential connective $\exists x : C.P(x)$.

```
Welcome to Coq 7.4 (Feb 2003)
Coq < Classical_Pred_Type.
Coq < Check not_all_ex_not.
not_all_ex_not :
          (U:Type; P:(U->Prop))~((n:U)(P n))->(EXT n:U|~(P n))
```

Therefore, instead of displaying the type of `not_all_ex_not` as

```
(U:Type; P:(U->Prop))~((n:U)(P n))->(EXT n:U|~(P n))        (1)
```

one would rather see

$$\forall \mathrm{U} : \mathsf{Type}; \mathrm{P} : \mathrm{U} \rightarrow \mathsf{Prop}. \neg(\forall n : U. (Pn)) \rightarrow (\exists\, \mathrm{n} : \mathrm{U}. \neg(Pn)) \tag{2}$$

Eventually, one would even hope to have the possibility of pretty printing this statement in natural language. For instance, instead of the raw Coq lemma:

```
Lemma Set_nrootIR :
  (n:nat)(lt (0) n)->(c:IR)(Zero [<=] c)
                 -> {x:IR & (Zero [<=] x) * (x[^]n [=] c)}
```

which depends heavily on the `Notation` enhancement mechanism, one prefers to see

Lemma 1. ***(Set_nrootIR)***

$$\forall n \in \mathbb{N}.0 < n \rightarrow \forall c \in \mathbb{R}.0 \leq c \rightarrow \exists x \in \mathbb{R}.0 \leq x \wedge x^n = c$$

or, even better,

Lemma 2. *For every positive integer n and non-negative real number c, there exists a non-negative real number x such that $x^n = c$.*

This last presentation is easier to read, hence it provides a better presentation for the formal statement within an article aimed for diffusion.

This raises several questions about pretty printing existing documents. Is the verbatim output, as shown in (1), sufficient to make it possible to display (2) or the two lemmas shown above? It should be clear that nothing is possible unless the text from (1) is *re-parsed* in order to grab enough information on the actual structure of this code fragment. The next question is: what is the best internal representation (abstract syntax) which is capable of providing the information required to make the various displays possible? And at what cost? Finally, what is the best way to generate the various styles and the formats?

The user expects other features as well. For instance, he might want to modify imported fragments to make them more readable or easier to understand. Thus, variable renaming, hypotheses reorganization, or other changes at the presentation level should be possible while keeping the formal development correct with respect to the underlying proof assistant. Such demands are reasonable since one cannot expect any automatic generation from a formal development to provide any article-ready text. As soon as interactive editing introduces changes with respect to the initial development, there is a possibility for introducing errors. It is these errors we would like to avoid with our tool.

2 Software Decisions

To build our tool, we have decided to use TeXmacs to write scientific articles and we have developed an extension for handling Coq input and output. The resulting system named TmCoq, implements the functionalities described in figure 1, where *editor* is TeXmacs. Actually, Postscript exportation is straightforward. For LaTeX exportation, we propose a specialized filter as explained in section 4.2. Re-exportation of the Coq script requires a special treatment in order to generate the right Coq syntax. We restrict ourselves to Coq developments for experimental purposes, with the long-term objective to build a tool adaptable to various theorem prover systems.

2.1 The scientific editor GNU TeXmacs

Quoting its documentation page [9], *"GNU TeXmacs is a free scientific text editor, which was both inspired by TeX and GNU Emacs. The editor allows you to write structured documents via a wysiwyg (what-you-see-is-what-you-get) and user friendly interface. New styles may be created by the user. The program implements high-quality typesetting algorithms and TeX fonts, which help you to produce professionally looking documents."*

We definitively want to offer the user a way to enrich their formal development with convenient documentation or informal explanations. Typically after

the automated importation step of a Coq script the user will do some interactive editing to prepare for dissemination of results. LaTeX exportation is satisfactory as a scientific document format. However, we must be able to recover the script to certify *a posteriori* the embedded code. Using LaTeX as the generic format for the script leads to a substantial loss of structure information. The resulting document is (enriched) text, while on the Coq side, a script is a program, where documentation is in comments. Therefore, the crucial distinction between the *active* parts of the script (the vernacular commands mainly) and the *inactive* parts (the interleaved comments-as-documentation fragments) is lost in the conversion processus if we use LaTeX as the generic format. Although this could be handled to some extent with the help of tricky annotations within the LaTeX end-of-line based comments, this is not satisfactory. The TeXmacs editor deals with structured documents, which is precisely what we want.

As far as presentation style is concern, the user expects to be allowed to customize the look, on file basis, on whole development basis, or even depending on whether the resulting document is about to be browsed or printed on paper. TeXmacs fulfills this criterion without requiring any special or additional code.

Moreover, TeXmacs also supports the Guile/Scheme extension language, so that it is easy to customize the interface and write extensions to the editor at the user level. Browsing is possible between distinct files as well as through the web. Automatic content generation includes indices, tables, figures, and bibliographies. TeXmacs documents are saved as structured text files using a simple notation system which helps automatic (machine-driven) search. Therefore, as a whole, TeXmacs presents many attractive features which make it an interesting candidate in the context of authoring scientific documents.

Communication with computer algebra systems Among our motivations, interactivity with the theorem prover assistant was also required. As such, TeXmacs provides a *session mode* that also offers the same combination of editing and command execution as Emacs does for many computer algebra systems, the *operating system shell*, and a *Guile/Scheme top-level*. Figure 2 presents a short session with Maple.

We see that the presentation facilities are automatically available for the output generated by Maple, while the session is allowed at any place within the document. Actually TeXmacs permits editing commands to be applied during the session, multiple interruptions, as well the interleaving of computations and explanations.

This session mode benefits to some extent from the standardization of mathematical notation and the relative simplicity of the concrete syntax found in Computer Algebra Systems (CAS). Also, it must be emphasized that LaTeX output is offered by Maple [1], Mathematica [2], and other CAS, which shows that the CAS must be adapted in order to gain richer output. Finally, note that the session mode is closer to the interaction the user gets with a calculator or the Unix shell than what is expected for developing proofs with the help of a Theorem Prover Assistant (TPA).

```
    |\^/|     Maple
._|\|   |/|_. Copyright by Waterloo Maple Inc
 \  MAPLE  /  All rights reserved. Maple is a registered trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.

Interface with TeXmacs by Christian Even (c) 2002.

Maple 1] f := x -> x^3+sin(x);
```

$$f := x \to x^3 + \sin(x)$$

Let us evaluate this function on (the default value of) π:

```
Maple 2] f(evalf(Pi));
```

$$31.00627669$$

```
Maple 3] f(Pi);
```

$$\pi^3$$

```
Maple 4] int(f(x),x);
```

$$\frac{1}{4}x^4 - \cos(x)$$

Fig. 2. A sample session with MAPLE

2.2 On the Coq side

To go further that COQDOC, our work proposes a solution to offer richer output formats to COQ. So we need to do more work than a simple lexical analysis. At least a syntax analysis is required. We will show that this is actually sufficient. However before this, we have to choose between writing another parser for our purposes, or benefiting from the existing one present in the COQ source. There are two major arguments to take into consideration. First, COQ is a research tool, still subject to changes, even with respect to its concrete syntax. Second, the syntax allowed at the formula level requires a very complex parser. Therefore, the necessary synchronization would already suggest that we develop our tool as part of the COQ source tree rather than another stand-alone program. But we can do better than just sharing the parser. Our code can also be used to interface COQ with PCOQ or any external tool that requires exchanging data with COQ.

Formulas Consider the following example, whichever notation one might choose, the internal structure for the above term `not_all_ex_not` is a tree (see figure 3), and the corresponding representation in $\TeX_{\text{MACS}}$ format is

∀{binder_list|P : U → Prop}. {arrowc|{N~_|~|(∀n : U. {appc|P n})}|({NEXT_:_|_| EXT |n|:||U|||¬(Pn)})}

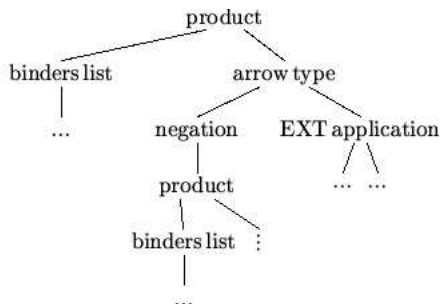


Fig. 3. Tree representation for `not_all_ex_not`

although partially unfolded. The binder_list, arrowc, and appc nodes should be obvious from above tree. The N˜_ and {NEXT_:_—_} come from extendible `Notation` command introduced in the last version of COQ that permits the kind of enhancement that (1) uses for the verbatim output. While this is not at all necessary in our tool (see RSA example below), at this stage of our development we decide to represent these nodes as special nodes in the generated structure.

Inductive definitions COQ provides specific outputs in many contexts, not only for formulas. Let us have a look at the `Inductive` vernacular command. The COQ standard library provides many examples. From `Datatypes` unit, the sum of two sets $A + B$ is inductively defined as

```
Inductive sum [A,B:Set] : Set
    := inl : A -> (sum A B)
     | inr : B -> (sum A B).
```

It is worth noting that the concrete syntax is usually chosen on the basis of

1. its ability to be read easily by the human reader,
2. and its ability to be read automatically, hence understood by the computer.

Come to think of it, both points would hold if we switched humans and computers as well! The important thing to keep in mind is that for both points it should be possible to infer from the concrete representation that an inductive definition is *structurally* made of a finite sequence of *named* items (`inl`, `inr` in this example) as constructors with respective type information. Also, the syntax [A,B: Set] means that sets `A` and `B` are *parameters* in the definition, the result being a set as the fragment ": Set" shows. Therefore, an inductive definition comes with its name, its parameters, the resulting type, and a finite list of constructors.

{Inductive|sum |A,B : Set |Set|{constructors|{constructor|inl} : {arrowc|A|(sum A B)}}}
{constructor|inr} : {arrowc|B|(sum A B)}}

So it suffices to grab these fields as part of the structure to be able to display the inductive definition in various styles. A straightforward representation in TEXMACS would be:

Inductive 3. (sum) *[A,B :* Set*] :* Set *:=*
*| **inl** : A → (sum A B)*
*| **inr** : B → (sum A B)*

A simple variation, closer to mathematical notation could be:

| **inl** : A → $A+B$

The same result can be obtained in LATEX from the corresponding source:

```
\Inductive{sum}{A,B : {\sort{Set}}}{{\sort{Set}}}{
  \begin{constructors}
    \constructor[inl] : ${\arrowc{A}{{\appc{sum A B}}}}$
    \contructor[inr]  : ${\arrowc{B}{{\appc{sum A B}}}}$
  \end{constructors}}
```

with the help of the appropriate macro definitions.

The more structure COQ is required to produce, the more freedom the user gets on their side to get the appropriate notation. As an example, one might want to present such inductive definition with the help of inference rules. From COQ user manual (section 4.5.1), one finds

```
(* Lists of elements of set A *)
Inductive list [A : Set] : Set :=
   nil : (list A) | cons : A -> (list A) -> (list A).

(* Length for such lists *)
Inductive Length [A:Set] : (list A) -> nat -> Prop :=
   Lnil : (Length A (nil A) O)
  | Lcons : (a:A)(l:(list A))(n:nat)
            (Length A l n)->(Length A (cons A a l) (S n)).
```

Depending on the context, presentations like

$$(\text{Lcons})\frac{a:A,l:\text{list}(A),n:\mathbb{N}\vdash(\text{Length}\,Aln)}{a:A,l:\text{list}(A),n:\mathbb{N}\vdash(\text{Length}\,A(a.l)(n+1))}$$

or

$$(\text{Lcons})\frac{\text{length}_A(l)=n}{\text{length}_A(a.l)=n+1}$$

when the context is clear enough, could be desired *without any change* in the underlying document. These representations raise no technical difficulty. Rather, they require more decomposition in the COQ structure representation output. In LATEX words, the constructor types for the case of inductive definitions could be prepared along the lines of this example:

```
\constructor[Lcons]
   \inferrule[a:A, l:(list A), n:nat]
                    {(Length A l n)}{(Length A (cons A a l) (S n)}
```

The section 4.5 addresses other places where such output questions have to be dealt with on the Coq side.

3 Utilization

The TmCoq system consists in several components:

- Coq which gives us parsing of scripts for free. Actually, our version is modified to allow for generation of TeXmacs format.
- A set of TeXmacs styles and macros for the rendering. The user is free to customize this package, the same way as LLNCS class does with respect to the standard LaTeX article class.
- A set of Scheme scripts to accomodate the LaTeX and Coq scripts exportation filters.

The last two components are available from the TeXmacs user interface.

For documentation purpose, our modified Coq distribution provides a new wrapper for the `coqtop` program, named `tmdoc`. A generic utilization consists in running the command:

```
# tmdoc MyScript.v
```

which results in the TeXmacs file `MyScript.tm`. Other output formats directly available from `tmdoc` are LaTeX, Scheme. However, as shown in figure (1), these formats together with HTML and Postscript are also available by editing `MyScript.tm`.

Let us illustrate further through two complete examples in the two following sections.

3.1 The Coq Standard Library

Let us have of look at the very beginning of the `theories/Init/Wf.v` file from the Coq standard library:

```
(*i $Id: uitp2003.tm,v 1.1 2003/05/27 11:30:57 paudebau Exp $ i*)

(** This module proves the validity of
     - well-founded recursion (also called course of values)
     - well-founded induction
    from a well-founded ordering on a given set *)
Require Logic.
Require LogicSyntax.

(** Well-founded induction principle on Prop *)

Variable A : Set.
Variable R : A -> A -> Prop.

 (** The accessibility predicate is defined to be non-informative *)

 Inductive  Acc : A -> Prop
    := Acc_intro : (x:A)((y:A)(R y x)->(Acc y))->(Acc x).

 Lemma Acc_inv : (x:A)(Acc x) -> (y:A)(R y x) -> (Acc y).
  NewDestruct 1; Trivial.
 Defined.
 (** the informative elimination : [let Acc_rec F = let rec wf x = F
x wf in wf] *)
```

Coq comments have the form `(*...*)`. The documentation fragments are identified by the special comment construction `(** ...*)` . This construction follows Coqdoc *almost text* conventions, which help writing in structured pieces of text without knowing too much of TEX or HTML syntax and more importantly in a independent syntax way. Therefore, to keep compatibility with existing documentation fragments in Coqdoc style, we have added to `tmdoc` another output format to make these tags understandable by $\text{T}_\text{E}\text{X}_{\text{MACS}}$. This comment is thus translated into

```
(** texmacs: This module proves the validity of
<\itemize>
<item> well-founded recursion (also called course of values)
<item> well-founded induction
</itemize>
   from a well-founded ordering on a given set  *)
```

One can see that we have extend the class of comments-as-documentation by the syntax `(** filter: ... *)` where `filter` ranges over `coqdoc`, `scheme`, `texmacs`, `latex` and defaults to `coqdoc` for compatibility's sake.

As for translation of the vernacular commands, our tool goes far beyond Coqdoc capabilities, but at the cost of requiring `coqtop` to perform both parsing and $\text{T}_\text{E}\text{X}_{\text{MACS}}$-aware structured output, as explained before.

This module proves the validity of

- well-founded recursion (also called course of values)
- well-founded induction

from a well-founded ordering on a given set

Well-founded induction principle on Prop

Variable A : Set
Variable R : A → A → Prop

The accessibility predicate is defined to be non-informative

Inductive 4. (Acc) *: A →* Prop *:=*
*| **Acc_intro** : ∀x:A. (∀y:A. (R y x) → (Acc y)) → (Acc x)*

Lemma 5. (Acc_inv)
∀x:A. (Acc x) → ∀y:A. (R y x) → (Acc y)

the informative elimination : `[let Acc_rec F = let rec wf x = F x wf in wf]`

Note that `Require` commands and proofs do not appear, although they have been translated and still remain present in the generated document. We provide flags that allow one to modify the material presented, which comes in handy for full script rendering or to show the mathematical contents of this file. Needless to say, $\mathrm{T_EX_{MACS}}$ offers many more options, either through the existing environment, macros writing, or GUILE scripting.

Following these brief explanations, the full COQ Standard Library is generated automatically by our documentation tool as a bunch of $\mathrm{T_EX_{MACS}}$ files which can be browsed within the editor. The blue items represent *hyperlinks*. Our contribution consists in a complete set of $\mathrm{T_EX_{MACS}}$ files accessible online from within the editor and browsable in the same way as the web documentation on the COQ site.

3.2 RSA Formalization

RSA is an asymmetric public key encryption algorithm which relies on prime number factorization. Its correctness has been formalized in various proof assistants including COQ by J.C. Almeida, and more recently by L. Thry using Fermat's theorem. The complete formalization consists of half a dozen COQ scripts available at [8].

In Binomials.v, one finds the well known Pascal statement on binomals:

```
Theorem exp_Pascal:
  (a, b, n : nat)
   (power (plus a b) n) =
   (sum_nm
    0 n
    [k : nat] (mult (binomial n k) (mult (power a (minus n k))
(power b k)))).
```

While the fragment does not contain any hint for presentation, the structure generated by `tmdoc` is rich enough to allow for user macro definitions so as to provide the following output.

Theorem 6. (exp_Pascal)
$\forall a,b,n{:}\mathit{nat}.\ (a+b)^n = \sum_0^n \lambda k{:}\,\mathrm{nat}.\,(\binom{n}{k} * (a^{(n-k)} * b^k))$

Observe, however, that the sum does not use the index `k` as one would expected. This is going to be a slight enhancement in the next stage of our development. In short, this requires that macro definition can inspect the structure of its arguments and proceed by case analysis.

For a complete presentation of our presentation contribution to this development, the required material is available from TmCoq web site [3]. Besides the same documentation interface as for the standard library, our contribution offers also a mechanism for automatically generating an article from the set of files. This is based on a Scheme script which is easily configurable.

4 Enhanced Features

One of our initial goals was to preserve the *invariant* that the script part of the document remains certifiable by the prover at any time. For this purpose, we have developed several functionalities described below, and based on the following remarks. In the introduction, we stated that a Coq script is a source program, where the inactive regions tagged as comments are good places for inserting documentation. The writing of an article from a formal development gives us another view. Everything is printable text, unless the macros or environments treat elements differently. In Knuth's view, *everything*, each character, could be active. However, for the casual user, the symbol '\' is commonly known as the first character of a document fragment which is going to be executed by the TeX engine. So the key idea is that the user is working in a *dual world* with respect to program source editing. Our tool transforms any *program source* file into a dual *document source* file, since TeXmacs is basically working in the same way as TeX.

Another property we get with the TeXmacs view of our document is that it is strongly *structured*. The visual structure of a text file as a sequence of paragraphs is of course represented in the source file where the document is saved. But this can be mixed with environments and other composition material in such a way

that a TEXMACS document is a tree with strings of characters as leaves. (At this point, it must be noted that TEXMACS does not use the XML and MATHML standards for file format on disk, this is ongoing work by its development team.)

This simple observation has significant consequences which are illustrated throughout the following sections.

4.1 Coq script (re)exportation

Assume we are dealing with a TEXMACS document generated from a COQ development. Among all the material present in this document, our tool ensures that any vernacular command is perfectly identified with the help of a specific macro (or environment). Thus, it is an easy task (thanks to TEXMACS's GUILE facility) to recover in the correct order all the vernacular commands among the rest of COQ-irrelevant material within the document. Once collected and converted into the actual COQ concrete syntax, these pieces of text compose the COQ script's active part that is thus extracted from the TEXMACS document, as shown in figure 1.

Everything else is considered documentation. To some extent, this could be inserted as comments into the script, but comments are thrown away by the COQ lexical analyzer. In order to collect the documentation provided in the source script, we had to change the lexer behavior, so that not all the comments are ignored. However, doing this in every place where comments may occur would result in a parser with unreasonable complexity. We decided on a model where documentation and code alternate, but no documentation is looked for within the vernacular commands. Documentation fragments may appear in many places within the TEXMACS representation of a given script. We ensure that they do not get lost through the exportation step by placing them in places where the (modified) COQ lexer is able to find them. However, we cannot guarantee that their exact position is preserved.

4.2 LATEX exportation

TEXMACS already supports LATEX exportation. However, the documents we are dealing are particular ones. First of all, they contain special material that is going to be interpreted through a default mechanism as LATEX macros in an inappropriate way. Without going into too much detail, macro expressibility is not exactly the same for LATEX and TEXMACS, which means that it is often better to build another macro rather than accepting the default solution. More importantly, the LATEX file exported from our document is a *particular view*. If it is meant for scientific publication, the user could choose to erase basic vernacular commands and proofs, while retaining only some of the definitions and other statements. Clearly for these cases, and probably for others, we want to have more control over the exportation process.

We have split the exportation in two steps, where the first one profits from the fact that we are able to identify the fragments specific to COQ among the whole document. This first step is designed as a GUILE script filter, which can

be rewritten by the user at will. Selecting the exported material can be locally tuned inside the document (source) by assigning values to specific variables locally, following the same idea as marking a fragment as math, **bold,** or *italic*. Afterwards, everything is put in the hands of the common LaTeX exportation filter. This solution offers both portability and enough expressiveness from the user point of view.

As far as the Coq source documentation is concerned, LaTeX exportation gives us this for free since the documentation is in the same *world*.

4.3 Coq-TeXmacs

Let us have a look at the `coq-tex` tool provided with Coq distribution. This is a stand-alone program which processes Coq phrases embedded in LaTeX files. This tool *greps* the LaTeX for Coq vernacular commands, identified by particular LaTeX environment tags (`coq_examples`, `coq_examples*`, `coq_eval`). The result is stored in a temporary script and sent to `coqtop` for evaluation. Depending on the surrounding tag, the command itself or the result are then re-inserted in place in the LaTeX file. As it stands, the environment tags offered with `coq-tex` gives the user the different possibilities based on two criteria: i) is the vernac command to be displayed and ii) is the result of its evaluation to be displayed? Whenever the answer to a question is positive, the appropriate text is printed verbatim at the appropriate place.

Assuming instead we start with a TeXmacs file, we can simulate the same behavior. However, we can do much more! The displayed material is no longer restricted to the raw verbatim format, nor the simple combination of two criteria. Moreover, since the source document is structured, there is no need for preparing an intermediate Coq script; whenever required, the corresponding answers can be directly inserted following the sent command. Thus, our `coq-texmacs` implements this functionality, based on the above exportation tools.

4.4 Certification in edition mode

In fact, we are very close to provide another functionality that we explain by making the comparison with a spelling checker. Inside TeXmacs the spelling checker is run the same way as inside Emacs. That is the editor selects each word one after the other and sends it to the spelling checker. The spelling checker is run in background. As far as spelling is concerned, the (atomic) item that has to be isolated in the buffer is a word, meaning a sequence of characters delimited by spaces or punctuation marks, depending on the underlying language. Assume now the spelling checker is Coq, words are to be replaced by vernacular commands. But we know how to isolate this kind of word, therefore making TeXmacs smart enough to send each vernacular command in the correct order to some Coq processus run in background. This mode offers a way to *certify* the (underlying) formal development inside the editor. Obviously, the expected behavior cannot be the same as for a spelling checker. Wherever an error is encountered,

this might inhibit the certification mechanism to continue. Actually, some dependency information between statements and their proofs will prove sufficient to allow to have a partial run of the certification process, leaving rejected (subtrees) pieces of our development marked as requiring further correction.

4.5 Interactivity

Since the user is working inside an interactive editor, it is highly desirable to offer on the fly certification, which requires that the editor keeps communicating with a Coq process. Towards this goal we have experimented with TEXMACS interactive mode with Coq. In section 2.1, we have shown that the editor TEXMACS offers a session mode with external programs. To make such communication possible with Coq, very little is needed as the following session with Coq shows.

A sample session with Coq Presented in figure 4.

Comments on this run From this simple example, we can make some useful observations. Up to now, our explanations were mainly concerned with formulas. Actually, exporting a full script does not raise any more difficult issues. But when interacting with Coq, it appears that the prover really decides the presentation in various contexts! The information displayed during a proof session, error messages, or answers to vernacular commands are controlled by the prover.

The most interesting issue concerns proof editing. Along the lines of the above session, one would expect the editor to allow the user to edit the proof tree, rather than accumulating steps in the buffer. This is possible in TEXMACS thanks to the Scheme extension language and provided through specific tree commands, together with the possibility for the document to be modified by the running program (as well as the user or the editor interface).

5 Conclusion and Perspectives

Our initial motivation was to offer the users more powerful solutions for documenting their Coq formal developments. Existing tools already offer useful means but lack the capability to render such mathematical or logical development in the usual language for scientific documents and none of them permit significant modifications or even final adjustments while preserving the fundamental property that the underlying formal document has been certified by the prover. In the first case, we conclude that this is only possible at the cost of working behind a dedicated parser. Due to complexity of the language dealt with by Coq and also motivated by the purpose to share our work and offer Coq various output formats, we decided to implement our documentation generator tmdoc as part of the Coq source distribution. At this stage, the rendering of mathematical formulas and the possibility to prepare a scientific article presenting the formal development was within the capabilities of the scientific editor TEXMACS.

The fact that this editor supports the GUILE/SCHEME extension language has also made it possible to handle the second case beyond our initial expectations. We provide COQ and LaTeX exportation functions both through the editor interface and as stand-alone scripts, which offer at the same a certification mechanism (on the COQ script) and the article production in the format commonly accepted by the scientific community.

Beyond these tools, our experimentation has shown very attractive perspectives. We want to achieve the functionality allowing the user to interact with the prover in a mode as transparent as possible. That means that the user will be allowed to write definitions, lemmas, and the other mathematical material in the syntax offered by the editor, leaving to the system all the conversions to be done in order to match the actual concrete syntax understood by the prover. Our view is that TeXmacs offers a nice interface for abstraction with respect to the prover. The interface can be adapted so that the user's syntax and commands do not depend on the prover. As such, this interface should offer a convenient support to teach beginners (students for instance) how to develop formal proofs. Even the certification mechanism could allow for machine-assisted verification of students' examinations. We consider our current experiments as an important step in this direction.

References

1. *The Maple analytical computation software*. `http://www.maplesoft.com/`.
2. *Mathematica: The Way the World Calculates*. `http://www.wolfram.com/products/mathematica/`.
3. Philippe Audebaud and Laurence Rideau. *Coq integration within TeXmacs*. Home page: `http://www-sop.inria.fr/lemme/Philippe.Audebaud/tmcoq/`.
4. Jean-Christophe Filliâtre. *Documentation tool for Coq*. Available at `http://www.lri.fr/~filliatr/\-coqdoc/`.
5. SRI Computer Science Laboratory. *The PVS Specification and Verification System*. `http://pvs.csl.sri.com/`.
6. Cornell Prl Automated Reasoning Project. *MetaPRL*. `http://cvs.metaprl.org:12000/metaprl/default.html`.
7. Coq Team. *The Coq Theorem Prover Assistant*. Available at `http://coq.inria.fr/`.
8. Laurent Théry. *A proof of correctness of RSA algorithm that relies on Fermat's little theorem*. Browsable online at `ftp://ftp-sop.inria.fr/lemme/Laurent.Thery/Rsa/index.html`.
9. Joris van der Hoeven. *The GNU TeXmacs free scientific text editor*. Available at `http://www.texmacs.org/`.

```
Welcome to Coq 7.4 (Feb 2003)
Interactive Session
```

```
Coq < Print all
   all  =  λA:Set; P:A → Prop. ∀x:A. (P  x)  :  ∀A:Set. (A → Prop) → Prop

Coq < Lemme 1. (foo)
∀a,b:Prop. a → b → a ∧ b
```

$$\frac{}{\forall a, b\!: \mathbf{Prop}.\, a \rightarrow b \rightarrow a \wedge b}$$

```
foo < Intros.
```

$$\frac{\begin{array}{lcl} a & : & \mathbf{Prop} \\ b & : & \mathbf{Prop} \\ H & : & a \\ H0 & : & b \end{array}}{a \wedge b}$$

```
foo < Split.
   2 sub-goal(s)
```

$$\frac{\vdots}{a}$$

```
   Subgoal 2 is b

foo < Exact H
```

$$\frac{\vdots}{b}$$

```
foo < Trivial
                         Subtree proved

foo < Save
   foo is defined

Coq < Print f
   Error : [6 - 7]  Error: f not a defined object

Coq < Print foo
   foo  =  λa,b:Prop; H:a; H0:b. <a, b> {H, H0}  :  ∀a,b:Prop. a → b → a ∧ b

Coq < Print bool

   Inductive 2. (bool) : Set :=
    | true : bool
    | false : bool
```

Fig. 4. A sample session with TmCoq

Visualizing Geometrical Statements with GeoView

Yves BERTOT, Frédérique GUILHOT and Loïc POTTIER

Lemme Project – INRIA
2004 route des lucioles - BP 93
FR - 06902 Sophia Antipolis
email: {bertot,fguilhot,pottier}@sophia.inria.fr

Abstract. We describe a tool that combines a theorem prover and an interface for dynamic geometry drawing to enhance man-machine interaction involving geometrical proofs. A general purpose theorem prover is used to formalize the statements and proofs and these are then visualized using an off-the-shelf drawing tool. The key component is an algorithm that computes the data needed to draw a construction from an algorithm that computes the data needed to draw a construction from the formal statement of the theorem. The paper includes some examples of output from our combined tool, called GeoView.

1 Introduction

General purpose theorem provers based on higher-order logic usually provide the possibility to develop proof interactively. Because of the interactive aspects, users need to develop competences in logic and reasoning. It is natural to think that this requirement makes this kind of theorem provers a good tool to help students learn mathematics.

We have studied this use of interactive theorem provers in the domain of geometry, concentrating on the courses available in the French high-school system. These courses encompass bi-dimensional euclidean geometry, with triangles, circles, basic transformations such as translations, rotations, homotheties and tri-dimensional geometry with parallelism, incidence and orthogonality problems between lines and planes. We have chosen to avoid analytic geometry, where reasoning becomes less important than computing.

All this work was performed using a graphical user-interface that made it possible to reconcile the requirements for a formal language (coming from a theorem prover) and the usual mathematical notations (more adapted for communication between humans, like teachers and students). In this context, it soon became clear that "a good drawing is better than a long explanation". We designed an extension of the graphical interface where drawings are automatically associated to mathematical formulas to show the meaning of these statements. Our main design decision was to reuse existing drawing tools. We have concentrated on tools already available to French math teachers, especially GeoplanJ [6], mainly because it is available with a GNU GPL license.

This paper is organized as follows: the first section is this introduction, in a second section, we describe the proof development tool that we use and the theory of geometrical facts that we have developed in this environment; in a third section, we summarize the functionalities of the independent drawing tool we integrated in the proof environment. The fourth section describes the crux of our contribution: an ordering algorithm to transform logical statements into figure construction sequences. The fifth section illustrates our results with a collection of significative examples. The sixth section studies related work and brings a few concluding remarks.

2 Pcoq and the geometry library

2.1 The Pcoq proof environment

Pcoq [1] is a user-interface for the general purpose theorem prover Coq [11] that manipulates all formulas and commands as structured data. This characteristic makes it easy to provide facilities that are hard to obtain in environments where the structure is not given. For instance, Pcoq makes it easy to attach special mathematical notations to some functions, thus making special notations for vectors, parallel and perpendicular predicates, angles possible. Examples of these notations will appear naturally in the figures below. An easy access to the structure of commands also makes it easy to attach sentences in natural language to geometrical statements.

Lemma **isocele_mediane_bissectrice:**
Let A, B, C and I be points.
If **A ≠ I,**
B ≠ C,
I is the midpoint of [B C],
and ABC is an isosceles triangle in A
then $(\overrightarrow{AB}, \overrightarrow{AI}) = (\overrightarrow{AI}, \overrightarrow{AC})$.

Fig. 1. Example of notations and sentences in natural language in Pcoq

As published in previous works [1, 9] natural language can also be used to give a readable form to the proofs built and verified by Coq. Combined with proof-by-pointing, all this gives a proof environment with good support for math teachers and pupils.

Easy access to structured data also plays a significant role in our experiment. The structured data corresponding to theorem or conjecture statements can easily be analyzed to generate figure texts that are transferred to GeoplanJ (described in section 3) for drawing and displaying. There remains some work to do to connect the statements to proper figure constructions, this work is described in section 4.

2.2 Our Geometry library for Coq

We have used Pcoq to develop a library of geometry theorems that encompasses the course as it is taught in French high-schools. This library is based on a large collection of axioms that is not chosen to be minimal but rather to correspond to the level of details that students are expected to understand and master.

Students are not supposed to know about algebraic notions as vector spaces, affine spaces, or metrics, but in spite of this, they are supposed to get acquainted with vector manipulations such adding, scalar multiplication, scalar product and so on. The general notion of barycentric computation cannot be used systematically, but the notion of barycenter is available.

Our axiomatization for affine space is very related to Marcel Berger's description of barycenters and the universal space in his book [3]. In our library, most operations are not given by a definition, but by the assumption that some function exists and satisfies one or two axioms.

We only enumerate here the different basic notions of plane geometry we can represent with a two-dimensional drawing:

- vectors, alignment, barycenter, midpoint, centroid, parallelism,
- orthogonality, orthocenter, orthogonal projection,
- euclidean distance, isosceles triangle, perpendicular bisector, circle,
- homothety, translation, reflection, rotation, direct similarity and composition of transformations.

We proved some classical theorems in plane geometry, in the same way as they are proved in the high-school courses. Among them, we can cite: Thales, Desargues, Pythagoras, Simson's line, Miquel, Euler's line, nine-point circle.

In this library, there are also chapters about trigonometry, complex numbers and three-dimensional geometry that we have not yet connected to a drawing tool.

3 GeoplanJ

3.1 The objects manipulated in GeoplanJ

The user creates mathematical objects in the plane equiped with a predefined but invisible coordinate system. Objects can belong to several classes. Some are *drawable*: points, straight or curved lines, and others are *undrawable*: real numbers, vectors, homotheties, translations, etc. Some objects are *free*, like arbitrary points and others are *bound*, like lines defined to pass through other points, points belonging to a line, midpoints, etc. Some objects are *fixed* in the sense that their description in the invisible coordinate system is completely given and some objects are *variable* in the sense that their position can still change.

All objects have a *value* describing their position in the actual drawing. The variable objects may move as a result from interaction by the user: their value may change. Thus, we can obtain several drawings for the same figure,

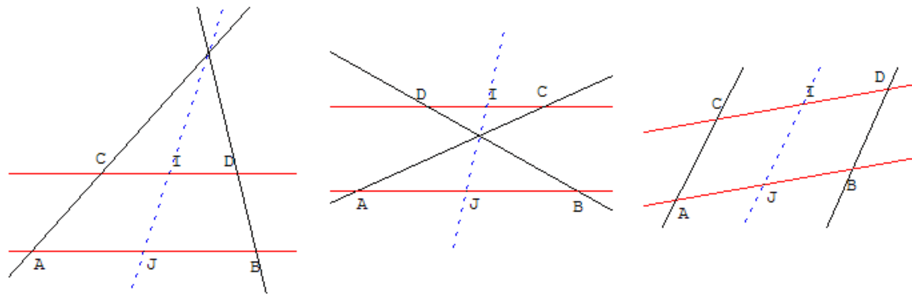


Fig. 2. Several drawings for the same figure

corresponding to different values for the variable objects. In figure 2, we show several "drawings" corresponding to the same figure.

The distinction between variable and free objects is meaningful: some objects may be variable and bound at the same time. For instance, a point may be bound to lie on a straight line but still be allowed to vary its position on the line.

A GeoplanJ figure is a collection of descriptions of points, including the constraints that bind them together. Every figure can be saved as a text where all objects are enumerated. It can also contain extra information, like color, line style, names. It is possible to ask GeoplanJ to draw a figure simply by giving it the text describing it.

3.2 Integrating the GeoplanJ software

GeoplanJ is a Java port of the software GeoplanW [7] and does not cover all the capabilities of the original software. It is initially intended for use as an applet, but since it is provided as free software under a very liberal license [5], we have been able to use it and modify it for our purposes. The main part of our proof development tool is also programmed in the Java language, so that including GeoplanJ in our software was an easy matter.

The GeoplanJ uses exactly the same textual format as GeoplanW to describe the figures, so that the figures generated in our experiment can be re-used and modified using the two pieces of software.

4 GeoView

Some predicates occuring in geometrical statements, like *collinear*, *cocyclic* and *isosceles* are multi-directional: when these predicates take $n+1$ arguments, given n of these it is possible to determine a constraint on the last one. However, some work needs to be done to decide in which order these predicates are used to build the figure associated to a statement. This work is done by the algorithm we present in this section.

4.1 Data analysis

The data corresponding to a theorem statement given as input is transformed in a tree of type: $H_1 \rightarrow \ldots \rightarrow H_n \rightarrow C$ where H_i can be interpreted as premises of the theorem and C as its conclusion.

Then we distinguish between *binding* and *non-binding* geometrical constraints (i.e. the constraint (collinear ABC) is a binding one but (not(collinear ABC)) is non-binding). The constraints which appear in the conclusion are redundant for construction: they are non-binding.

We list all the binding constraints and the points which appear in them. From this list, an algorithm detailed in 4.2 determines a figure text with GeoplanJ syntax.

4.2 Construction of set of points from a set of geometrical constraints

Given a finite set of points in the real plane, and a finite set of geometrical constraints on these points, we describe a simple algorithm that may produce a construction of these points, suitable to draw a picture of these points satifying the geometrical contraints.

Geometrical constraints

A *geometrical constraint* is for instance a constraint that can be formulated as a set of polynomial equalities on the coordinates of points: saying that points A,B, C are collinear is a valid constraint, because it is equivalent to one polynomial equality: $(x_C - x_A)(y_B - y_A) = (y_C - y_A)(x_B - x_A)$. A geometrical constraint is written as $(CP_1 \ldots P_n)$, where C is the constraint and $P_1 \ldots P_n$ are the points.

We will mainly restrict ourselves to constraints such that, for some choice of $n - 1$ points among $P_1 \ldots P_n$, we can build the remaining point by simple geometric constructions from the others. For instance, for the constraint (collinear ABC), we can choose two points freely, the third being constrained to lie on the straight line defined by the two others.

To a constraint C we associate its type $T(C)$ which describes the degree of freedom of the points. If the remaining point P_i (called the *linked point*) is completely determined, the constraint will be said of type 2 (i.e. the two coordinates of the point are determined). If P_i can vary on a fixed point curve (circle or straight line in general), the constraint is of type 1.

Examples:

- (collinear ABC): means that A, B and C are collinear .T(collinear) $= 1$
- (midpoint ABI): means that I is the midpoint of $[AB]$. T(midpoint) $= 2$.

Constructions of points

To build the linked point of a constraint we need to consider its position in the constraint, and we may have to build intermediate objects.

For instance, in the constraint (isosceles ABC) which means $AB = AC$, to build the point A, we build the perpendicular bisector of $[BC]$, but to build the point B we build the circle of center A with radius $[AC]$.

If a point appears in two constraints of type 1, we build the intersection of two geometrical objects (straight lines, circles, straight line and circle).

From constraints to construction

We can now describe how to build a set of points $P_1, \dots, P_m$ satifying a set of constraints

$(C_1 Q_{11} \dots Q_{1r_1}), \dots, (C_n Q_{n1} \dots Q_{nr_n})$ where $\forall ij, Q_{ij} \in \{P_1, \dots, P_m\}$.

Let M be a matrix with n lines and m columns. An entry M_{ij} of the matrix M will say how the point P_j is used by the constraint C_i :

- $M_{ij} = -1$: P_j does not appear in the constraint C_i,
- $M_{ij} = 0$: P_j is not a linked point in the constraint C_i,
- $M_{ij} = 1$: P_j is a linked point in the constraint C_i of type 1,
- $M_{ij} = 2$: P_j is a linked point in the constraint C_i of type 2.

Our goal is to find a matrix verifying the conditions:

1. $\forall ij, M_{ij} \in \{-1, 0, 1, 2\}$,
2. $\forall ij, M_{ij} = -1 \Leftrightarrow P_j$ does not appear in the constraint C_i,
3. $\forall ij, M_{ij} > 0 \Rightarrow P_j$ appears in the constraint C_i of type M_{ij},
4. each line of M has exactly one strictly positive entry,
5. $\forall j, \sum_i \sup(0, M_{ij}) \leqslant 2$,
6. the relation $\prec$ on points $P_1, ..., P_m$ defined by

$$a \prec b \Leftrightarrow \exists i, M_{ia} = 0 \,\text{and}\, M_{\text{ib}} > 0$$

   is such that its transitive closure $\preccurlyeq$ is antisymetric.

The condition 4 means that each constraint will build a new point (possibly not completely determined if the constraint is of type 1).

The condition 5 means that a point is built by one or two constraints, and if it is built by two it is the intersection of two objects.

The condition 6 means that the construction does not loop: we do not use a built point to build an older one.

The algorithm to find such a matrix is now simple to describe:

i. we first build a matrix M verifying conditions 1, 2, 3 and 4, where for each line the positive entry is left-most, which is easy.
ii. we enumerate all matrix verifying conditions 1, 2, 3 and 4, simply by lexicographically shifting the positive entry of each line to the right. We stop as soon as the matrix verifies conditions 5 and 6, which are easy to check. Otherwise, we fail.

If this algorithm does not fail, we can use the produced matrix M directly to obtain the effective construction of the points: the minimal points for the relation $\lll$ (which is a partial order) are taken are free points in the plane. From this set of points, called S_0, with the constraints, we can build a new set of points S_1 (which is the minimal points for $\lll$ in $(\{P_1, \ldots, P_n\} - S_0)$. And so on.

Example

Let us take the following points and constraints:

C_1: (collinear A B H)

C_2 :(ortho H C A B), i.e. the straight lines (HC) and (AB) are perpendicular.

C_3: (circle A B C), i.e. C is on the circle with diameter $[AB]$.

We have following types :

$T(\text{collinear}) = 1$

$T(\text{ortho}) = 1$

$T(\text{circle}) = 1$

Let $P_1, \ldots, P_n = A, B, C, H$.

The initial matrix, verifying conditions 1, 2, 3 and 4, is:

M	A	B	C	H
collinear	1	0	-1	0
ortho	1	0	0	0
circle	1	0	0	-1

It does not verify condition 5: the column A is linked by three constraints.

In lexicographic order, the next matrix verifying conditions 1, 2, 3 and 4 is:

M	A	B	C	H
collinear	1	0	-1	0
ortho	1	0	0	0
circle	0	1	0	-1

It verifies condition 5, but not condition 6: we have $B \prec A \prec B$

Continuing, we get the first next matrix that verifies conditions 5 and 6:

M	A	B	C	H
collinear	1	0	-1	0
ortho	0	0	1	0
circle	0	0	1	-1

We have then $B \prec A, H \prec A, A \prec C$.

The construction is then:

1. Take B and H two free points in the plane.
2. Take A as a free point on the line (BH).
3. Take C as the intersection between the line orthogonal to (AB) and containing H and the circle with diameter $[AB]$.

Generic configurations, limitations and improvements

The whole previous discussion on free points (in the plane, on a straight line or a circle) only holds generically, i.e. outside a set of particular cases (for instance, when two points are equal, radius of circles are two small, etc). But this set is of finite mesure in $\mathbb{R}^n$, so when choosing free points at random, we have probability one to be out of these degenerated situations.

This method is not complete. From a set of solvable geometrical constraints, it can fail to give a construction. The reason is that we work on non-linear constraints on points: this kind of constraints define a good degree of freedom not on points but on coordinates, or on more complexe structures (determinants of projective points for instance).

Being not complete, the method however has the advantage to be simple and fast, and it still fails very rarely.

To describe the degree of freedom of points, the type of a constraint is not always sufficient. In further work, we will replace the type T of a constraint C by a list of types LT which describe the degree of freedom of points depending of their position in the constraint. For instance, for the constraint (reflection $ABMN$) which means that N is the image point of M by reflection in line (AB), the point N is completely determined by the position of the three other points A, B and M, we will say that the constraint is of type 2 in position 4. But to build B, we cannot choose freely the three points M, N and A, we will say that the constraint is of type 0 in position 2. Hense we have $LT(\text{reflection}) = (0, 0, 2, 2)$.

5 A few illustrating examples

After analyzing the constraint matrix, a figure text is transferred to GeoplanJ so that the free points appear in green and the conclusion in red in the generated figure. The user can actually move the free points and the figure evolves accordingly.

5.1 Nine-point circle theorem

We proved in Coq the nine-point circle theorem, also called Euler's circle theorem and Feuerbach's circle theorem, using vector calculus, oriented angles of vectors and homothety properties as in mathematics high-school. Given a triangle, this circle passes through the three midpoints of the sides, the feet of the perpendicular to each side passing through the opposite point and also through the midpoints of the segments which join the vertices and the orthocenter. We also proved that the center of this circle is the midpoint of the segment joining the circumcenter O and the orthocenter H of the triangle and that the centroid G of the triangle lies on line (OH) which is called Euler's line.

In the following illustration, we show this theorem statement as it appears in Pcoq and below the corresponding figure generated by Geoview where A, B and C are the only free points.

```
Theorem cercle_neuf_points:
 Let A, B, C, A', B', C', O, G, H, I, J, K, L, H1, H2 and H3 be points.
 If ABC is a triangle,
  A' is the midpoint of [BC],
  B' is the midpoint of [AC],
  C' is the midpoint of [AB],
  G is the centroid of triangle ABC,
  O is the circumcenter of triangle ABC,
  H is the orthocenter of triangle ABC,
  I is the midpoint of [OH],
  J is the midpoint of [AH],
  K is the midpoint of [BH],
  L is the midpoint of [CH],
  H1 is the orthogonal projection of A on line (BC),
  H2 is the orthogonal projection of B on line (CA),
 and H3 is the orthogonal projection of C on line (AB)
 then
 J is on the circumcircle of triangle A'B'C' and H1 is on the circumcircle of triangle A'B'C' and
 K is on the circumcircle of triangle A'B'C' and H2 is on the circumcircle of triangle A'B'C' and
 L is on the circumcircle of triangle A'B'C' and H3 is on the circumcircle of triangle A'B'C'.
```

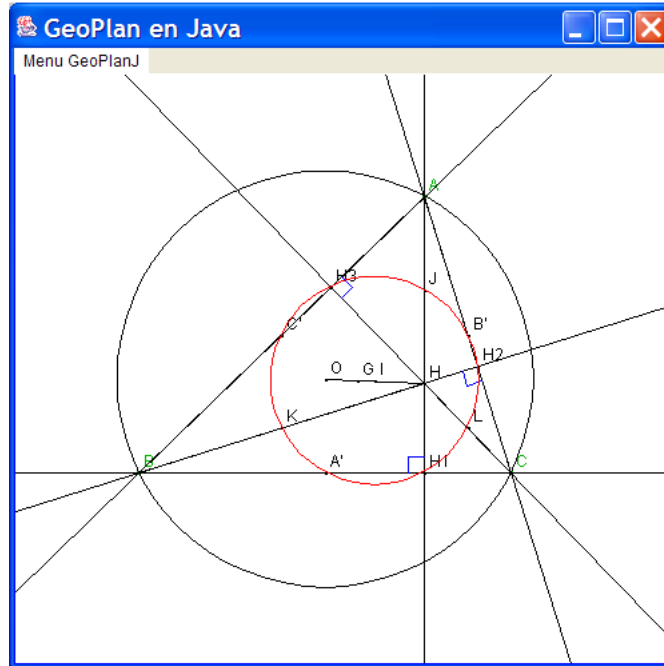


Fig. 3. Nine point circle theorem: statement in Pcoq and figure generated by GeoView

5.2 Simson's line theorem

We proved in Coq Simson's line theorem, using oriented angles of vectors. This theorem states that: given a triangle and M a point in the plane, the three feet of the perpendiculars from M to the sides of the triangle are collinear if and only if M is on the circumcircle of the triangle.

In this example, the conclusion of the theorem is an equivalence of two propositions. With Geoview, we consider the first one as a premise and the second one as the conclusion. In the following illustration, we show this theorem statement as it appears in Pcoq and the corresponding figure generated by Geoview.

Theorem **Simson_line:**
Let A, B, C, M, P, Q and R be points.
If **ABC is a triangle,**
BCM is a triangle,
ABM is a triangle,
ACM is a triangle,
P is the orthogonal projection of M on line (BC),
Q is the orthogonal projection of M on line (AC),
and R is the orthogonal projection of M on line (AB)
then **M is on the circumcircle of triangle ABC if and only if P, Q and R are collinear.**

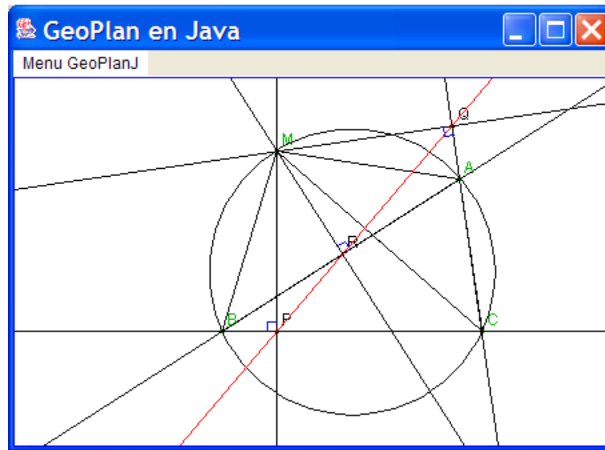


Fig. 4. Simson's theorem: statement in Pcoq and figure generated by GeoView

5.3 Disjunctions

Disjunctions that occur among the premises of a theorem imply that actually several figures must be considered, one for each disjunct.

Disjunctions that occur inside the conclusion of a theorem correspond to the fact that several configurations may occur for the same figure. This is already illustrated in figure 2 where the lines (AC), (BD), and (IJ) either intersect or are parallel.

5.4 Representing real numbers

Real numbers that occur in vector expressions or geometric transformations such as homotheties or rotations are displayed as points on a fixed straight line, displayed on top of the drawing, as it appears in figure 6. Moving the point with the mouse makes it possible to change the value of the real number, other geometrical objects that depend on this real move accordingly.

5.5 Existential quantification

As we already said, the constraints that appear in the conclusion of a statement are usually non-binding. If the conclusion is an existential quantification, then this introduces binding constraints, even though they occur in the conclusion.

A good example is given by the composition of a translation by vector $\overrightarrow{AA'}$ and a non-trivial homothety of center I and ratio k. The theorem states that there exists a point J that is bound with the points I, A and A' and the ratio k of the homothety.

The theorem statement appears in Pcoq as follows:

```
Lemma composee_translation_homothetie_exists:
 Let k be a real.
 Let I, A and A' be points.
 If  k ≠ 1,
  and  k ≠ 0
 then
 exists a point J such that : for all points B, B' and B1,
 If  B1 is the image point of B by the translation by vector AA',
  and  B' is the image point of B1 by the homothety of center I and ratio k
 then  B' is the image point of B by the homothety of center J and ratio k.
```

```
Lemma composee_translation_homothetie_exists:
 ∀k : R.
 ∀I, A, A' : PO.
 k ≠ 1 =>
  k ≠ 0 =>
   ∃J : PO .
   ∀B, B', B1 : PO.
   B1 = (translation A A' B) => B' = (homothetie k I B1) => B' = (homothetie k J B).
```

Fig. 5. Two different ways of displaying the same theorem statement in Pcoq

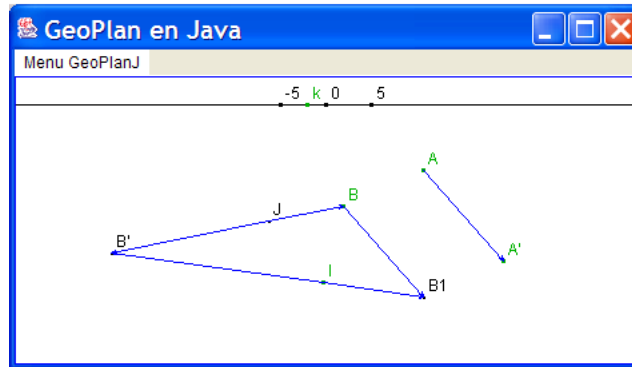


Fig. 6. The corresponding figure generated by GeoView.

The corresponding figure can be drawn as follows:

In this figure, J moves whenever I, k and A and A' do, but does not move when B does. The point J is independent from B as is expressed by the nesting of universal or existential quantifiers in the theorem statement.

6 Concluding remarks

6.1 Related work

Our work is at the boundary between two fields: the first is concerned with interactive drawings, while the second is concerned with reasoning tools. Some experiments to connect these two fields deserve our attention.

Drawing tools for geometric constructions abound, many Java applets, that can be used in a web-browser can easily be found by searching on the world-wide web. For example, we can cite the work of D.E. Joyce that we could find and test easily [8].

In our work, we have concentrated on tools already available to French math teachers such as Cabri-Géomètre [12] (which we have not used) and GeoPlanW and GeoSpaceW [7], and especially a Java Port of GeoPlanW, known as GeoPlanJ [6]. The data created with our tool can then be re-used and modified by the teachers for presentations in other contexts.

Reasoning tools for geometry can also be organized in two categories. A first category uses analytic methods to reduce geometric problems to algebraic problems and powerful algebraic tools (such as gröbner bases) to solve the latter. A second category relies on a prover, often a first order prover to deduce facts from a collection of axioms describing geometry. In the first category, we can cite the work of Shang-Ching Chou [4] and in the second category the work of Dominique Py [10] or the Leibniz laboratory managed by Nicolas Balacheff [2]. Our work is closer to the second category, especially since the point of view that

tools are a teaching aid prevails in all these experiments. In particular, Baghera [13] also provides ways to display the drawings associated to exercises but it takes care of organizing the classroom or the schoolbag from the perspective of the teacher or the pupil.

6.2 Conclusion

It should be clear to the reader that the experiment we have described in this paper is easily adapted to other proof or drawing tools. The geometry library itself should easily be described with any other proof tool, even the axiomatization itself can be changed. The experiment only requires to map basic notions from the geometry library to the basic notions of the drawing tool: points, alignments, cocyclic constraints, etc. For instance, Cabri-Geometre is one of the most popular tools in use in the French school system and it seems easy to adapt our work to this drawing tool.

The model of interaction that is proposed in this experiment uses the drawing tool only as an output device: formulas are taken from the written form and transformed into figure descriptions. It would be interesting to consider a reverse interaction scheme, where a figure would be described using the mouse by interacting with the drawing tool and then translated into a statement or a proof step. However, it seems the expressive power of the drawing tool is limited, since undrawable objects are difficult to describe naturally. Also, the drawing makes it difficult to distinguish between premises and conclusions of logical statements.

Another perspective is to adapt this work to three-dimensional geometry. For now, there are few drawing tools available to consider the third dimension and to ease our experiments it matters that the tool should be given in a publicly available, open source manner. Anyway, representing three dimensional objects on the two-dimensional screen poses difficulties. Planes in the three-dimensional space are difficult to render in an efficient way. The usual trick is to represent a plane by a parallelogram included in that plane, but then two intersecting planes are not guaranteed to exhibit an intersection in any drawing. Conversely, two straight lines that do not intersect may appear to be intersecting when the image is rendered on a flat screen.

Acknowledgements

We would like to thank Frédéric Kotecki, the author of GeoplanJ for his collaboration and Farid Latrech who worked on GeoView in July 2002.

References

1. A. Amerkad, Y. Bertot, L. Pottier, and L. Rideau. Mathematics and Proof Presentation in Pcoq. In *Proceedings of Workshop Proof Transformation and Presentation and Proof Complexities in connection with IJCAR 2001*, Siena, Italy, June 2001.
2. N. Balacheff, http://www-didactique.imag.fr/.

3. M. Berger. *Geometry I*, chapter "barycenters; the universal space", pages 67–83. Springer, 1987.
4. S. Chou, X. Gao, and J. Zhang. *Machine proofs in geometry: automated production of readable proofs for geometry problems*. World scientific, 1994.
5. Free Software Foundation, http://www.fsf.org/.
6. GeoplanJ, http://mapage.noos.fr/fkotecki/geoplanj.html.
7. GeoplanW, http://www2.cnam.fr/creem/geoplanw/geoplanw.htm.
8. D.E. Joyce, http://aleph0.clarku.edu/ djoyce/java/geometry/geometry.html.
9. H. Naciri and L. Rideau. Formal Mathematical Proof Explanations in Natural Language Using MathML: An Application to Proofs in Arabic. In *MathML International Conference: Hickory Ridge Conference Center, Chicago, IL, USA, June 28–30*, 2002.
10. D. Py. *Environnements Interactifs d'Apprentissage et démonstration en géométrie*. Habilitation à diriger des recherches, Université de Rennes 1, 2001.
11. The Coq Development Team. The Coq proof assistant: Reference manual: Version 7.2. Technical Report RT-0255, INRIA, February 2002.
12. Jill Vincent. *Exploring 2-dimensional space with Cabri Geometry II*. Mathematical association of victoria.
13. C. Webber and S. Pesty. Emergent diagnosis via coalition formation. In Garijo F., editor, *8th Iberoamerican Conference on Artificial Intelligence (IBERAMIA 2002)*, pages 755–764, Spain, November 2002. LNAI 2527, Springer Verlag.

Rendering Tree Proofs in Box-and-Line Form

Richard Bornat[1]

School of Computing Science, Middlesex University. R.Bornat@mdx.ac.uk

Abstract. Some recent improvements in the design of Jape, a proof calculator, are described. Jape's internal data structure is a sequent tree, but it now supports an accurate box-and-line treatment of natural deduction. Changes to its internal workings, to its tactic language and to user interaction are described.

1 Background

In 1998 the proof calculator Jape[3, 8, 9, 1] was assessed by an educational researcher, as part of a project on visualisation[2]. It had been in existence for about eight years, and under constant development. Nevertheless, the effect of evaluation was salutary – as evaluations so often are – and led to several significant changes in the external and internal machinery of Jape.

First, it became clear that novices, unlike experts, don't benefit from the freedom to defer choices. The sort of 'exploration' offered to those learning formal proof is quite difficult enough when restricted to choices about which rule to apply and when. Jape's use of unknowns to defer the specialisation of a logical step (usually an unknown argument in a substitution) is a piece of meta-machinery which derails a novice's fragile understanding. Couple this with an awkward use of text substitution as a device to smuggle additional arguments past Jape's point-and-click primitivism, and few undergraduates could get beyond simple propositional examples. In one heartrending videotaped session a student wailed that she didn't know why 'it always does this' when she used a $\forall$ elimination rule (figure 1).[1]

So much meta-logical internal machinery is showing that it is no wonder she despaired. An entire section of the manual distributed to first years was described procedures for eliminating unknowns once they had appeared, and a subsection was dedicated to strategies for avoiding them in the first place. In this case the manual suggested that you should use multiple text selections and unification to replace *_c1* by c, or text-select c to provide it as a second argument in the rule step. In my course on GUI design I taught students that help text is an admission of defeat. Manuals are an even worse dereliction of duty, and I finally learnt to take my own medicine. To solve the problem I had to implement multiple antecedent selection in the GUI module, interpret it in Jape's tactic

[1] Figures 1 and 2 have been generated from a modern version of Jape. They suggest what was seen in 1998, but the reality was far worse.

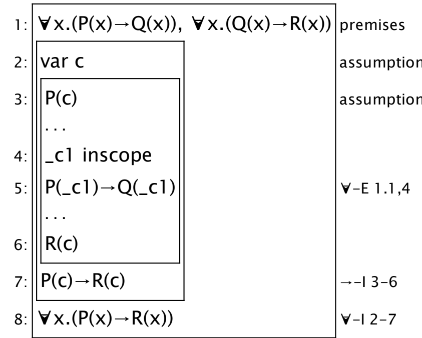


Fig. 1. An incomplete ∀ elimination step

language, and modify the logic so that certain steps became unnecessary (which last I wanted to do anyway, for other reasons). The benefit was that proof search strategies became more explicable in terms of the logic, and explanations had less to do with the difficulties of instructing Jape to do the right thing.

Second, those videotaped students showed that my advice to novices to read all Jape's error messages as "bang!!" wasn't working. They tried to understand what Jape said to them, and were understandably demoralised when they couldn't. Some error messages are generated by simple slops: many of us, for example, sometimes ask for an elimination step when we wanted an introduction step; the mistake is encouraged by the partical identification of introduction steps as 'backward' and elimination steps as 'forward'. In one session a pair of novices were confronted with an error dialogue box which looked something like figure 2. The dialog box used language which they had barely encountered to describe internal Jape objects that they couldn't see on the screen. It was designed to suit the needs of an expert logic encoder working in the sequent calculus, or able to envisage the internal workings of Jape behind a box-and-line proof display. To solve this problem I had to write lots of special tactics that tease apart possible causes of error and construct individual error messages appropriate to each different situation. In particular it was necessary to reorganise the menus, so that error messages which talked about backward and forward steps related to visible elements of the interface. And it meant more changes to the logic, particularly to the treatment of negation and contradiction, to remove some unnecessary causes of error.

These two changes – eliminating incomplete steps, improving error reporting – addressed the needs of novices learning a particular logic. They took a considerable amount of particularly focussed effort. Unfortunately, other logic encodings couldn't immediately benefit without similar expenditure of effort (Jape's tactic language doesn't allows much code reuse!).

The third significant problem affected every user, because it was about the correct display of box-and-line proofs. Simply, Jape didn't display true box-and-line proofs, because you couldn't always call on earlier in-

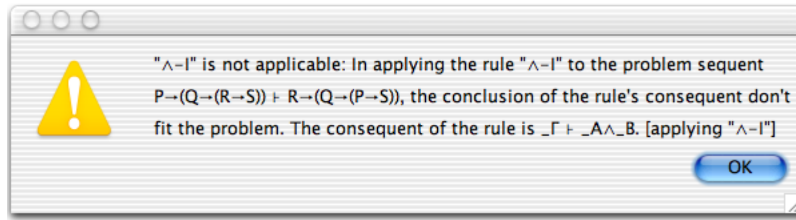


Fig. 2. An inappropriate error message

scope lines in later deductions. There were certain rules of thumb which an encoding expert could use to reduce the problem, like making elimination steps before introduction steps, but they didn't always work and in any case imposed a significant and unnecessary planning load. My Jape publicity claimed that you could choose what to do and where to do it, but it didn't say that innocent choices could have unjustifiable consequences. The problem needed fixing: it was already a problem for me as an expert user, and it would bite the novices once the other obstacles had been cleaned away. The complexities of the fix are described below. The final problem was gestural. Jape made no visible distinction between 'hypothesis' and 'conclusion' formula selections, although to command Jape it's necessary to observe and exploit the distinction. Jape now displays these diffferent selections differently, and this facilitates true forward steps (see below) which allow a line of the proof to be used either as hypothesis or as conclusion.

2 Rendering Tree Proofs in Box-and-Line Style

Jape's basic box-and-line proof rendering mechanisms are dealt with elsewhere [6, 8], and I give merely a summary. Sequent tree proofs make large displays, partly because the same left formulae are repeated over and over again in the tree and partly because of branching of the tree. For example, figure 3 is a tree proof in a natural-deduction style sequent calculus (introduction and elimination rules plus an identity rule $\Gamma, A \vdash A$, here called `hyp`). The same proof can be shown far more compactly as figure 5. Jape's rendering algorithm converts the tree into the compact box-and-line display in four stages.

2.1 Stage 1: Linearise

Jape renders each tree node as a sequence of boxes and lines. The linear rendering of figure 3 is shown in figure 4. If a node has more left formulae in its consequent than its parent has (the root and the node above it in figure 3, for example) then it is rendered as a box starting with the extra left formulae labelled as assumptions or premises. Otherwise it generates a sequence of the renderings of its subtrees, followed by a line containing

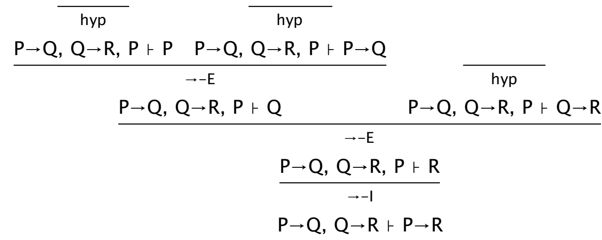


Fig. 3. A wide tree proof

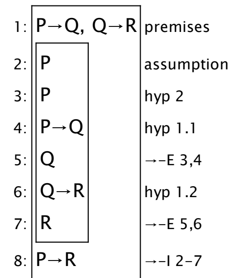


Fig. 4. A linearised proof

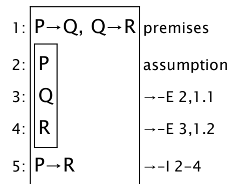


Fig. 5. A compact box proof

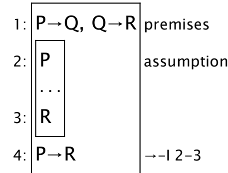


Fig. 6. First step of the proof

the consequent formula labelled with the name of the rule that generated the node. Tips generate a single unlabelled line.
Thus the root of figure 3 generates a box starting on line 1 of figure fig:boxproofwithhyp, followed by the rendering of its subtree (lines 2-7), followed on line 8 by the root formula labelled with →-I. The root's subtree is also rendered as a box: line 2 followed by its left subtree (lines 3-5), its right subtree (line 6) and line 7 labelled with →-E. Each consequent line refers to the subtrees either as *i*-*j*, the first and last line numbers of a subtree box, or *j*, the last line number of a sequence of lines. Rules which match a left formula (in this logic the only such rule is `hyp`) refer to an occurrence in the premises or assumptions. All these effects can be seen in the example, and it's easy to reconstruct the tree from its linearised form.

2.2 Stage 2: Hide Identity Lines

Lines 3, 4 and 6 of figure 3 are no more than indirections. Jape normally elides these lines, replacing references to them by references to the premise or assumption to which they refer. This converts figure 4 into figure 5. Most identity steps are carried out behind the scenes anyway, and they aren't really part of the natural deduction narrative, so the elision makes a lot of sense from the prover's point of view. After this stage it's only a little more difficult to reconstruct the tree from the linearised elided form.

2.3 Stage 3: Hide Cut Steps to Allow Forward Reasoning

Figure 6 shows how a user might make the first step in a proof. The tree produced is just the last three lines of figure 3. If the next step is backward, producing the lower five lines of figure 3, then the display is as shown in figure 7. But most novices would prefer to make a forward step, producing figure 8. The forward step is indicated by selecting a hypothesis formula – in this case $P \rightarrow Q$ – and invoking an appropriate rule.

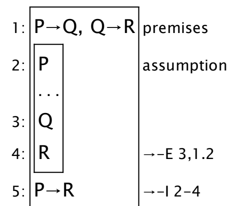


Fig. 7. A backward step from figure 6

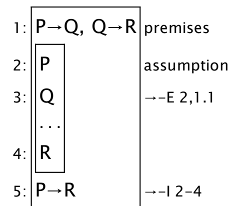


Fig. 8. A forward step from figure 6

Forward steps generate a new formula which can be called upon by deductions below it – that is, a new hypothesis formula. That's only possible if the new formula is a left formula. In a natural deduction style

logic, the only way to introduce a new left formula is to use the cut rule $(\Gamma \vdash A; \Gamma, A \vdash B) \Rightarrow \Gamma \vdash B$. Behind the scenes, figure 8 uses the tree of figure 9; the stage 2 rendering of that tree is shown in figure 10. To produce figure 8, Jape conflates the first antecedent of the cut with the assumption line, the conclusion of the second antecedent with the conclusion of the cut, elides the box, and adjusts the labelling of lines.

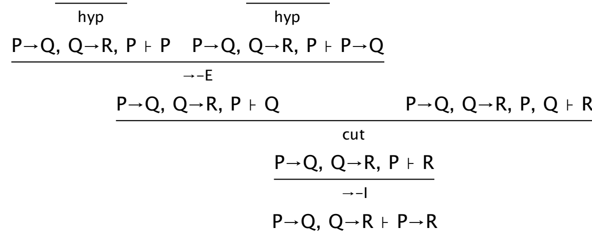


Fig. 9. A tree with a forward step

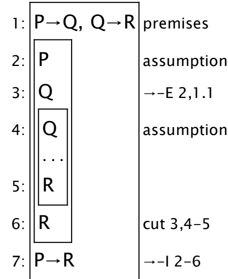


Fig. 10. A linearised cut step

2.4 Stage 4: Deal with Transitive Operations

In dealing with chains of reasoning such as $A = B$, $B = C$, ... $Y = Z$, it's possible to treat reflexivity $A = A$ and symmetry $A = B \Rightarrow B = A$l like logical identity and treat transitivity $(A = B; B = C) \Rightarrow A = C$ as a kind of cut, generating a sequence of lines $A =, = B, = C, ..., = Z$. Jape's rendering mechanism can do this, but it isn't directly relevant to this discussion.

2.5 What's Wrong with This Mechanism?

The rendering mechanism goes wrong in stage 1. In a box-and-line proof it's possible to use any previous line (boxes permitting) when making a

step: that is, the underlying structure is a DAG. In a tree, on the other hand, you can't refer to sibling subtrees. When a tree node is rendered, therefore, although the lines representing the deductions of subtree n come before the lines of subtree $n+1$, they must all be inaccessible to the lines of the sibling subtree below. In multiplicative (context-splitting) logics the inaccuracy is even worse, because the left formulae – shown as premises and assumptions – accessible from one subtree will not be the same as those accessible from another subtree, and the box-and-line rendering doesn't show the difference.

The problem is illustrated, and there is a hint of its possible solution, in the trees of figure 11 and 12. These different trees each generate the same box-and-line display, shown in figure 13. To generate figure 11 the first step is $\wedge$ introduction, and then a new left formula Q is generated in the left subtree by a forward $\rightarrow$ elimination step, closing that subtree. If the proof is to be completed, the same forward step will have to be duplicated in the second subtree. To generate figure 12 the first step is a forward $\rightarrow$ elimination, which generates a left formula Q usable by the whole of the proof. The next step is $\wedge$ introduction; the left formula of the cut closes the left subtree, and it's also available to the right subtree.

hyp hyp
P, P→Q, Q→R ⊢ P   P, P→Q, Q→R ⊢ P→Q
→-E hyp
P, P→Q, Q→R ⊢ Q   P, P→Q, Q→R, Q ⊢ Q
cut
P, P→Q, Q→R ⊢ Q   P, P→Q, Q→R ⊢ R
∧-I
P, P→Q, Q→R ⊢ Q∧R

Fig. 11. Intro then elim

hyp hyp hyp
P, P→Q, Q→R ⊢ P   P, P→Q, Q→R ⊢ P→Q   P, P→Q, Q→R, Q ⊢ Q   P, P→Q, Q→R, Q ⊢ R
→-E ∧-I
P, P→Q, Q→R ⊢ Q   P, P→Q, Q→R, Q ⊢ Q∧R
cut
P, P→Q, Q→R ⊢ Q∧R

Fig. 12. Elim then intro

Figure 13 isn't quite as ambiguous as it seems. From the earliest days of Jape, Bernard Sufrin and I used a greying-out mechanism to show which left formulae were visible in a box-and-line proof but not accessible. If the tree behind figure 13 is figure 11, for example, then selecting line 3 will grey out line 2; if the tree is 12, line 2 will be undimmed. The manual advised users to select a position in the tree by clicking on an

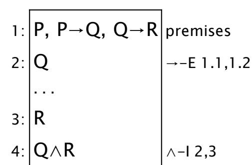


Fig. 13. An ambiguous display

active consequent – a formula below a line of three dots – and observe which formulae above the consequent remained visibly active and which were greyed out. If the results broke the rules of box-and-line proofs then novices perhaps thought it no more surprising than anything else Jape did, experts perhaps could explain it by thinking of the underlying working of the tree, and probably everybody was too grateful for the display of proof context to complain about its inaccuracies. At any rate I don't recall much criticism of our treatment.

This was the state of Jape for three or four years: inaccurately rendered proofs, and imperfectly executed forward steps. If the problem suited it, and if you were prepared, as most logical novice users are, to make all possible forward steps before any backward step,[2] then the misrepresentation of tree proofs seemed much less of a problem.

The problem isn't caused or made worse by the introduction of forward steps into a backwards-reasoning sequent-tree calculator. On the contrary, forward steps provide a way to increase efficiency, by choosing to do as much as possible by forward reasoning early on in the proof. Jape's deficiencies are perceived as inefficiency of the user's proof strategy caused by poor planning: that is, Jape is imposing a planning load on its users.

3 True Forward Steps

The first step towards a solution of the problem is to deal properly with forward steps. If a forward step always produces a tree like figure 12, never like figure 11, whichever order the forward and backward steps are undertaken, then at least one part of the problem is solved. In the latest versions of Jape, forward steps can insert a cut node at the lowest point possible in the tree. This means that, for the first time, users can make steps which are not directed towards a conclusion-tip. For example, the partial proof of figure 14 can evolve into figure 15, inserting a hypothesis formula directly below the line it derives from. This is a true forward step.

To make Jape take the step required some new internal mechanisms. Previously, the tactic which implemented forward steps was

[2] Novices believe with great fervour, no matter how they are instructed, in the unique efficacy of forward reasoning. They resist backward reasoning with the same conviction. I've concluded that they think backwards steps are cheating.

```
1: (E→F)∧(E→G)  premise
2: | E          assumption
   | ...
3: | F∧G
4: E→(F∧G)      → intro 2-3
```

Fig. 14. Before forward step

```
1: (E→F)∧(E→G)  premise
2: E→F          ∧ elim 1
3: | E          assumption
   | ...
4: | F∧G
5: E→(F∧G)      → intro 3-4
```

Fig. 15. After forward step

```
TACTIC ForwardCut (n,rule)
    SEQ cut
        (LETGOALPATH G (WITHARGSEL rule)
                       (GOALPATH (SUBGOAL G n))
                       (WITHHYPSEL hyp)
                       (GOALPATH G)
                       NEXTGOAL)
```

– apply cut; record the current position (left subgoal of the cut) by binding its tree path to G; make the user's proof step; move to the nth subgoal of the rule step; using the user's hypothesis selection, close with a selected left formula; move back to position G; look for the next unclosed tip using right-to-left recursive traversal of the tree.

The new tactic is superficially similar:

```
TACTIC ForwardCut (n,rule)
    CUTIN (LETGOALPATH G (WITHARGSEL Rule)
                         (GOALPATH (SUBGOAL G n))
                         (WITHHYPSEL hyp)
                         (GOALPATH G))
          )
```

Like all Jape tactics, `ForwardCut` executes at a position in a proof tree. To begin, the tree position is defined by the user's formula selections: a conclusion selection defines a tip; a hypothesis selection defines the lowest point in the tree at which it is present as a left formula; if there's more than one selected position then the greying-out mechanisms guarantee that they are all on the same path in the tree, and Jape uses the highest. Then the `CUTIN` operation, which can only be used if the logic has a suitable cut rule and has been specified in `ADDITIVELEFT` style, looks for the lowest point in the tree which has exactly the same left formulae as the selected position. It breaks the tree there, tops the break with a cut node using the old top section as its right subtree, and augments every left context in the old top section with the cut formula, generating new provisos where necessary.[3] Then it executes its argument tactic at the left subtree of the new cut node, and finally it returns to the point in the tree at which it was originally applied. The effect is to splice a forward step into the tree.

[3] In Jape's rule notation `FRESH c` requires that the name `c` can't appear free in left or right formulae of the consequent of a step which uses a rule. Nodes at which `FRESH` has been applied are marked, and `CUTIN` must add a proviso for the formula it is inserting, each time it encounters such a node.

The complication is navigation: Jape tactics navigate by storing tree paths (LETGOALPATH and GOALPATH, for example, in the tactics above), which are invalidated if a new node is inserted into the tree somewhere along the path. Rather than attempt to search for and correct all stored paths, I implemented a path-following algorithm which ignores inserted cut nodes unless they are explicitly mentioned in the path formula. (Easy to say, horrible to implement.)

4 Building a DAG

Forward steps are only half of the problem. Multi-antecedent backward steps are the other half. In Jape's rendering mechanism, a formula on a previous line is only accessible if it represents an immediate antecedent or it represents a left formula (a hypothesis, in the language we use to talk about Jape proofs). The lines and boxes describing subtree n aren't direct antecedents of anything in subtree $n+1$. There is nothing else for it: they have to be hypotheses; that is, they have to be introduced by cut steps. And then, by induction, it's clear that *every* backward step has to generate new hypotheses. This apparently absurd situation is the only solution to the problem that fits the bill.
The way to generate new hypotheses, low enough down in the tree that they can be accessed by lines which might need them, is of course to use CUTIN. The tactic which is behind the menu entry for ∧ intro, for example, is

```
TACTIC "∧ intro backward" IS SEQ "∧ intro" fstep fstep
TACTIC fstep IS
    ALT (ANY (MATCH hyp))
        (trueforward SKIP)
MACRO trueforward(tac) IS
    LETGOAL _A (CUTIN (LETGOAL _B (UNIFY _A _B)
                                  tac))
               (ANY (MATCH hyp))
```

– apply the ∧ intro rule, then apply the fstep tactic to each of its antecedents. The fstep tactic checks that the tip it is applied to can't be closed by an identity step (if it can, there's no need to introduce a new left formula) and then applies trueforward; that tactic records the consequent of the tip it is applied to (LETGOAL); then runs CUTIN, unifying the new cut formula with the original consequent and applying the argument tactic (in this case simply SKIP to the left subtree of the cut; finally, back at the original position in the tree, closes the tip with an identity step.
The effect, since this mechanism is used for *every* backward step, is to close every backward step by identity with a left formula generated lower down in the tree. All the identity steps will be invisible in the box-and-line display, because they are elided in stage 2 of the rendering algorithm; the lower position will be above the positions which inserted hypotheses visible to the backward step; stage 3 of the rendering algorithm, applied recursively, exposes hypotheses in lowest-first order; it follows that the lines of backward steps, if fstep is uniformly applied at every stage,

appear in the correct order. In effect, the antecedents of a backward step are inserted below and push the lines above them upwards, rather than growing the proof above the step itself.
The total effect, if we think of cuts as augmenting an environment and rule steps as generating nodes, is to generate a DAG. A cut which introduces a left formula A in effect produces a tree which is described by the formula “let $line = treeA$ in $tree$” where $tree$ can refer as many times as necessary to $line$. This is a description of the spanning tree of a DAG. It seems inside out, and it is odd to use the proof tree in this way, but it does work!

5 Pointing to Formulae

In a sequent, left formulae are separated from right formulae by the turnstile, so that a selection-click on a left formula is spatially and obviously distinct from one on a right formula. Jape’s tactic language was built from the first to allow discrimination between left (`LETHYP`) and right selections (`LETCONC`). When Bernard Sufrin and I first implemented box-and-line display of tree proofs, we retained the same distinction, since left formulae were labelled premise or assumption, and right formulae of tips were unlabelled and below a line of dots.
Before the developments described in this paper, stage 3 of the rendering mechanism hid two sorts of cut steps: one where the cut formula was a hypothesis alone, with no lines above it that could use it as a target conclusion; the other where the cut formula was a conclusion, with no lines below it that could call upon it as hypothesis. All formulae in the display, whether generated normally or by forward steps, could then be classified as hypothesis or conclusion, and clicks on them treated appropriately as left or right selections. This meant that the rendering mechanism couldn’t deal with raw cut steps – ones which generated a formula that was a possible bridge between hypotheses and conclusion, but not yet connected to either – because it wasn’t possible to classify the occurrence of the cut formula as purely hypothesis or purely conclusion.
Under this rendering mechanism active conclusion formulae appeared below a line of dots, and hypotheses were those formulae above an active conclusion which didn’t grey out when the conclusion was selected. All very well, provided that you clicked on a conclusion before every step, but novices didn’t do that very often (remember, they prefer to reason forwards). Some more emphatic distinction seemed necessary, and even before implementing forward steps I began to make hypothesis selections – boxes open downwards – visibly distinct from conclusion selections – boxes open upwards. Also, following Bernard Sufrin’s lead from his implementation of the Jape GUI module for Linux, I made the selection boxes red as shown in figure 16.
True forward steps made it essential to deal with raw cuts. In an $\wedge$ intro step, for example, the left-hand formula can be used as a conclusion to be proved from the lines above it, as in figure 17, or a hypothesis to be used in the lines below it, as in figure 18. As these examples show, Jape’s rendering mechanism can now hide all cut steps, and interprets

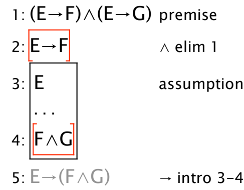


Fig. 16. Directed selection boxes

clicks on the top half of an ambiguously-usable cut formula as conclusion selection (upward-pointing), on the bottom half as hypothesis selection (downward-pointing). In either case the closed end of the selection is drawn with a dashed line, intended to suggest that it's only a temporary assignment of rôle.

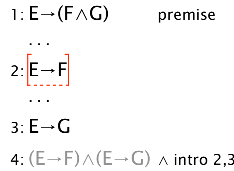


Fig. 17. A cut-conclusion

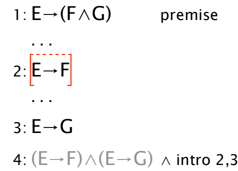


Fig. 18. A cut-hypothesis

The selection gesture now makes more demand on a prover's motor control than before, but the main issue is that the hypothesis/conclusion distinction sometimes has to be chosen by the prover. That means that novices now have to understand the distinction, where previously they might have been able to ignore it and blunder through. I think that perhaps this is a good thing.

6 Assessment

The four mechanisms discussed above made quite a difference to the usability of the logic encoding that I presented to my first-year undergraduate class, but they haven't been documented or widely publicised nor yet independently assessed, and so it's difficult to assess their eventual impact. The selection gesture improvement seems like a no-brainer, though a scientific evaluation would be needed to be certain. The modified logic encoding, with its improved error messages, its elimination of incomplete steps, its more rational set of logical rules and its correct display of box-and-line proofs, seems to be a success with novices and is unlikely to be completely misconceived, though my experience of educational evaluation warns me that there must be many points of detail which could be improved.

What is not to be celebrated is the means of implementing all this. Throughout the early development of Jape[4, 12, 5, 13, 7], Bernard Sufrin and I made great play of the notion that what is on the screen is an interpretation of what is in the machine, eliding unnecessary parts and showing tidied up versions of others. What was in the machine was supposed to be transparently an encoding of a logic; what appeared was supposed to fit the demands of proof development and display. It was and is the case that sequent trees are the easiest and most transparent way to represent formal logical proofs. It's easy to understand how to implement operations like applying a rule at a tip, or deleting the subtree(s) above a node. It's easy to implement a mechanism to enforce a proviso like "c must not occur in this sequent", and thus implement privacy conditions (eigenvariable conditions) on quantifier rules.

We also aimed to provide a lightweight way of encoding logics in Jape's rule and tactic notation. The developments described here seem to have subverted that aim: in my latest encoding of natural deduction the encoding of logical rules is a mere 45 lines and 1.4K characters; the tactics and definitions which support interaction via clicks and menu commands are 800 lines and 37K where, prior to improvement, they had been only 54 lines and 2K.

Much of the extra encoding volume is tactic programming that constructs error messages for particular situations. I'm confident that that is an inevitable cost of attempting to generate messages by interpreting user's intentions, though no doubt it would be worth searching for more transparent and economical representations of the search. The rest of the increase is caused by the complexity of the representation of forward steps, the need to use them everywhere, and the fact that they have to be carefully crafted if they aren't to break down with inexplicable error messages, to generate duplicate hypothesis lines, or loop for ever. In the particular case of natural deduction, which was the second example I tackled, it wasn't very difficult to get it right but then (a) I designed the mechanisms, so I understood them, and (b) in the other example, which was a treatment of Hoare logic and had to deal with large displays, I never quite got it right. It's hard to avoid the conclusion that for Jape tactic programming, this may already be a bridge too far.

Least satisfactory of all is the use of cuts to implement a DAG. The tree mechanism, which was until recently quite clearly behind the implementation of box-and-line proofs, is now little more than the scaffold on which quite a different structure is built. Worst of all, DAG generation is imperfectly implemented: in particular, the treatment of transitive reasoning is not integrated into it, and it does not support the next obvious step, which is moving lines around in a box-and-line display to produce a more pleasing arrangement.

7 Where Next?

Jape was designed as a sequent tree calculator, because Bernard Sufrin and I understood the syntax and semantics of trees. We knew we could get the tree part right and then work from a solid engineering basis. For a

long time this architecture was a success, and the difficulty of rendering a tree proof in box-and-line form was comparable to the difficulty of rendering a tree proof directly.[4] Now it's a very much more complicated business, and the complexity extends outside the module that actually renders the picture into the tactic interpreter (via `CUTIN`), the prooftree navigator (because of the need to carefully interpret paths once a cut step has been inserted), and worst of all the logic encoding (via **`fstep`** and the like).

One way forward is obvious. All that the complexity is accomplishing is the preservation of an illusion that what you see – a box-and-line proof, i.e. a DAG – is really there. If the illusion were reality, it would surely be much easier to implement the insertion of a line, which is all that `CUTIN` does. And if the DAG were the underlying representation, new mechanisms can be imagined which are impossibly daunting today: users could rearrange the lines of a displayed proof, for example, and it might be possible to satisfy undergraduate students' desire for a mechanism which would take in their pencil-and-paper proofs and comment on their correctness.

The drawback of such a development would be that Jape would no longer embrace forms of logic that don't fit DAGs. That would be a pity, but two lessons I have learnt from building Jape are to recognise the beauty of natural-deduction-style logics on the one hand, and to realise the range of logical invention and hence the difficulty of squeezing many useful logics even into Jape's sequent idiom. BI[11] and its cousin separation logic[10], for example, have trees in their left contexts, which is hard to mimic in Jape let alone implement effectively. Jape could usefully retreat from the hubris of universal aims and be made to support natural deduction reasoning even better than it does now.

All that is perhaps. It would be a significant effort to comprehend and implement the semantics of box-and-line proofs so that all the mechanisms Jape currently supports are accurately translated, and so that there is room for development.[5] It would be a still larger effort to re-design and re-implement all the parts of Jape that work on the tree. Maybe the recent port to Windows[1] is my last push, and Jape's swansong.

Acknowledgements

Bernard Sufrin and I designed and implemented Jape together for several years. His architectural insight and crucial early design decisions have been essential underpinning for everything I have managed to achieve since he moved on to other things. His continued advice helps me implement along the narrow path.

[4] So far Jape has a very simplistic recursive rendering of tree displays. I did manage to snuggle little subtrees under the branches of big ones to reduce width in some cases, but most tree displays are large, short and wide. It's possible to imagine a renderer which can draw a sequent as a block rather than a single line, which would make it possible to draw trees with a nicer aspect ratio, but they would still be large.

[5] Actually, since I began to write this paper, and in particular since I've noted the analogy between DAGs and let formulae, it doesn't seem so hard any more. Oh dear!

My colleagues at Queen Mary College, especially Adam Eppendahl, Mike Samuels, Graham White, Paul Taylor (who advised me on redesigning the logic rules) and Keith Clarke, all of whom discussed Jape's design and implementation with me in the ten years that it was developed and deployed there, helped me to see some of the ways that I pushed Jape forward.
James Aczel and his colleagues at the Open University evaluated what I thought was quite a good version of Jape. His exposure of its deficiencies, and his insights into the causes of students' difficulties, drove all the developments discussed above. The anonymous students who agreed to be videotaped did him and me and their successors a great service.
The mistakes in Jape, and the complexity of the current implementation, are nowadays all my fault (especially because I ignored so much wise advice over the years).

References

1. The jape web site. http://www.jape.org.uk. Latest versions of Jape for various platforms, including the encoding of natural deduction discussed in this paper.
2. J. C. Aczel, P. Fung, R. Bornat, M. Oliver, T. OShea, and B. Sufrin. Using computers to learn logic: undergraduates experiences. In G. Cumming, T. Okamoto, and L. Gomez, editors, *Advanced Research in Computers and Communications in Education: Proceedings of the 7th International Conference on Computers in Education*, Amsterdam, 1999. IOS Press.
3. R. Bornat and B. Sufrin. Jape: A calculator for animating proof-on-paper. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 412–415, Berlin, July 13–17 1997. Springer.
4. Richard Bornat. User interface principles for theorem provers. invited talk at UITP 95, Glasgow, 1995.
5. Richard Bornat and Bernard Sufrin. Jape's quiet interface. presented at UITP96, York, 1996.
6. Richard Bornat and Bernard Sufrin. Displaying sequent-calculus proofs in natural-deduction style: experience with the jape proof calculator. presented at International Workshop on Proof Transformation and Presentation, Dagstuhl, 1997.
7. Richard Bornat and Bernard Sufrin. Using gestures to disambiguate search and unification. presented at UITP 98, Eindhoven, 1998.
8. Richard Bornat and Bernard Sufrin. Animating formal proof at the surface: The Jape proof calculator. *The Computer Journal*, 42(3):177–192, 1999.
9. Richard Bornat and Bernard Sufrin. A minimal graphical user interface for the jape proof calculator. *Formal Aspects of Computing*, 11(3):244–271, 1999.
10. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 14–26, London, January 2001.

11. D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.
12. Bernard Sufrin and Richard Bornat. User interfaces for generic proof assistants. part i: Interpreting gestures. presented at UITP96, York, 1996.
13. Bernard Sufrin and Richard Bornat. User interfaces for generic proof assistants. part 2: Displaying proofs. presented at UITP 98, Eindhoven, 1998.

Interactive disproof

Richard Bornat

School of Computing Science, Middlesex University. R.Bornat@mdx.ac.uk

Abstract. The proof calculator Jape has been extended to allow disproof, using Kripke forcing semantics. Users draw graphs to depict a situation, and Jape uses subformula colouring to signal the status of the situation and its components. The mechanism hasn't been scientifically evaluated, but it has been heavily tested in use. Some deficiencies are noted.

1 Introduction

Jape[3, 5, 6, 1] is a proof calculator. It was devised for use in education, specifically for early education in formal proof, and for running lightweight experiments with novel logics in formal methods research. It has a claim to be novel in that the calculator is designed to support the user interface rather than the other way around.

Recently I had experience of using Jape as the foundation of a first course in logic (until then my experience had been limited to advising others and observing[2] the results). Jape's formal proof mechanisms had been overhauled, as reported elsewhere[4], to increase its usability. In the first year of use, those improvements certainly seemed to be helpful. Stung by the criticisms of colleagues, however, I decided also to try to teach some model theory. Since constructive proof is so much more straightforward than classical proof (it really is!), my course focussed on constructive proof. It seemed only right that I should tackle constructive model theory.

The most fruitful way, I imagine, of incorporating model theory into a course on proof is to treat it as a means of disproof. If you can show a counter-example in the model, and if the logic is sound (ours is, but proof of soundness is not a part of the course) then a formal proof is impossible. You can even use a failed search for a constructive proof to guide the search for a constructive counter-example. You can have fun, in those cases where a classical proof and a constructive counter-example exist, comparing proof and counter-example of contentious examples.

All of this was done, for one year, on blackboard and on paper. The contrast between the ease with which most students learnt from Jape-supported proof search, where the machine was accepted fairly readily as an accurate and impartial arbiter in difficult cases, and the pain they experienced with paper-and-pencil counter-examples, where the only arbiters were fallible teachers, was stark. Constructive counter-models are graphical constructs with very simple rules: it was plain that machine support might be provided, and so I attempted it.

2 A scant introduction to Kripke semantics

Classical logic is based on the notion of truth. A properly formed assertion is either true or false, independently of our understanding of it. The corresponding 'method of truth tables', enumerating all possible states of the support for an assertion, is familiar to most computer scientists, at least for the propositional case and for the purposes of designing hardware. Kripke's constructive model using possible worlds[8, 7], though graphical and very accessible, is less well known. I therefore give a very brief introduction, at the level of engineering mathematics rather than foundation.

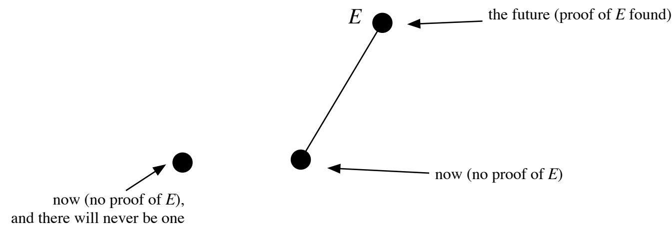


Fig. 1. Two alternative views of the present

Constructive logic is based on the notion of proof. A properly formed assertion can be proved or disproved or there can be no evidence either way. (It's not a three-valued classical logic – it's odder than that[7].) In contrast with classical logic, then, it's reasonable to consider a distinction between 'now' and 'the future': we might have no proof of assertion E now, but one might be found tomorrow, as recently was the case for Fermat's Last Theorem, and as we might hope for Goldbach's Conjecture. In the semantics the two possible situations can be shown by alternative diagrams, as in figure 1. The first (left-hand) diagram describes a situation in which there is no proof of E, and none will ever be found. The second (right-hand) world describes a situation in which there is no proof of E now, but one will be found. We don't know, when we don't have a proof of E, which of these worlds we are living in, so it might well be the second. Since constructivists interpret $\neg E$ as 'there will never be a proof of E', the second world denies both E and $\neg E$, and thus the classical law of excluded middle. On the other hand, a diagram with just a single world and no future development, like the left-hand one above, corresponds to the classical model.
Jape supports the drawing of these possible-world diagrams. The natural deduction encoding described here uses the definitions of figure 2 where $w \geq w'$ means you can travel from w to w' by following upward lines in a diagram. The glory of this collection is the $\rightarrow$, which captures beautifully the notion that I am forced to accept $A \rightarrow B$ if, however things might develop, whenever I'm forced to accept A then I'm also forced to accept B.

$w \vDash A \wedge B$ iff $w \vDash A$ and $w \vDash B$
$w \vDash A \vee B$ iff $w \vDash A$ or $w \vDash B$
$w \vDash A \rightarrow B$ iff for every $w' \geq w$, if $w' \vDash A$ then $w' \vDash B$
$w \vDash \neg A$ iff for no $w' \geq w, w' \vDash A$
$w \vDash \forall x.P(x)$ iff for every $w' \geq w$ and every i, if $w' \vDash$ actual i then $w' \vDash P(i)$
$w \vDash \exists x.P(x)$ iff for some $i, w \vDash$ actual i and $w \vDash P(i)$

Fig. 2. Constructive forcing semantics of natural deduction

3 Drawing counter-examples

The assertion $(E \rightarrow F) \vee (F \rightarrow E)$ has no constructive proof. An attempt to prove it using constructive rules in Jape gets stuck quite quickly, as illustrated in figure 3. There is still a constructive rule which is applicable – from contradiction conclude F – but it's pointless to try it because you can't generate a contradiction from E. The experienced counter-example generator recognises from the structure of the stuck proof that the counter-example will probably involve a world at which there is a proof of E but no proof of F.

1:	E	assumption
	...	
2:	F	
3:	E→F	→ intro 1–2
4:	(E→F)∨(F→E)	∨ intro 3

Fig. 3. A stuck constructive proof

Choosing Disprove in the menus splits the proof window into two panes and shows a minimal semantica situation, as illustrated in figure 4. The red-ringed blob in the upper pane is the currently-selected world; the assertion below it is coloured to show the status of its components in that world. Atomic assertions E and F are black, to show that they are not forced. The arrow in $E \rightarrow F$, like the brackets surrounding that subformula, is coloured violet to show that the subformula is forced – trivially, because E appears nowhere in the diagram. The arrow and brackets of $F \rightarrow E$ are coloured similarly, for similar reasons. The disjunction connective, between the brackets, is violet because $F \rightarrow E$ is forced, and the whole assertion is underlined in violet to make clear that it is forced. A counter-example demonstrates that an assertion does not always hold by showing a situation in which all the premises are forced but the conclusion is not. This is not yet a counter-example.
A new world can be created by option-dragging the base world, producing figure 5. Note that the new world makes no change in the colouring,

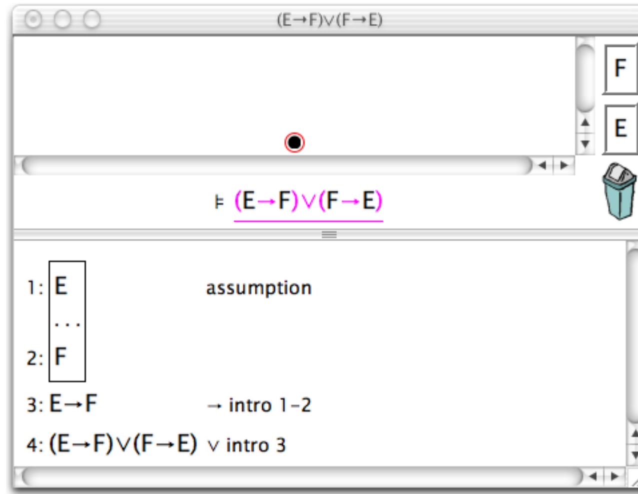


Fig. 4. The beginning of a disproof attempt

because it introduces no future developments. Then E can be forced at the new world by dragging the E tile from the right-hand side of the window and dropping it on the new world, producing figure 6. Now the colouring changes: $E \rightarrow F$ is no longer forced, because there is a reachable world at which E is forced and F is not. But $F \rightarrow E$, and therefore the whole assertion, is still forced for the same reason as before.

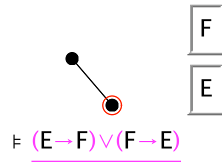


Fig. 5. An extra world

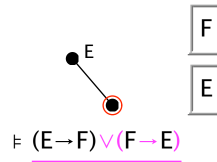


Fig. 6. A labelled extra world

At this point the experienced counter-example generator knows just what to do to complete the disproof: do for F what was just done for E. That's justified because the first proof step picked out the left-hand half of the disjunction, and we might as easily have picked the right, giving figure 7. Previously, to make a new world I dragged a new one out of the base world. That is the right thing to do again, but it is no longer the only thing to do, so I illustrate the effect of alternative choices. First, the easy way and a correct move: dragging a new world out of the base and dropping F onto it completes a disproof, signalled in figure 8 by the fact that the assertion is now *not* underlined.

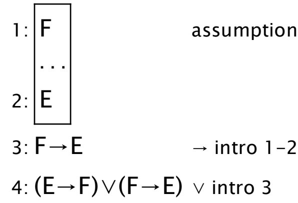


Fig. 7. Stuck again

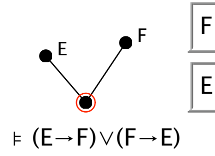


Fig. 8. The completed disproof

The other, more laboured, route to a counter-example starts by extending the E world as in figure 9. Jape enforces monotonicity when the new world is created by including E in the new world. Note that the colouring and underlining of the assertion is unchanged from the single-E situation. Then when F is dropped on the new world to give figure 10 we find that $F \rightarrow E$ is still forced, because in the top world, the only world in which F is forced, so is E. This situation isn't a counter-example.

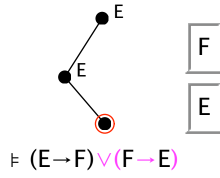


Fig. 9. Extending the upper world

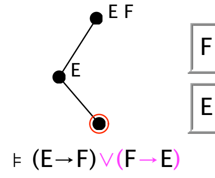


Fig. 10. Not a counter-example

It's possible to go forward in at least two ways. One way is to undo back to the two-world situation and do the right thing from there (Jape applies undo and redo to the last pane – proof or disproof – that you clicked in, rather than using an overall interaction history). Alternatively, you can modify the current situation by deleting parts of it. Monotonicity stopped us making a world with just F in it, so first I decide to get rid of the link I just made (but not the world): press on the link from the E world to the E, F world and drag it to the swing-bin in the bottom right of the disproof pane (the line stays attached to its endpoints and drags rubber-band fashion till it is dropped in the bin, which lights up, as it should, to receive the undesired object – but it's hard to capture that moment!), giving figure 11. Option-dragging the base world onto the isolated E, F world makes a new link, giving figure 12. Finally, throwing the undesired E in the bin takes us back to the counter-example of figure 8.

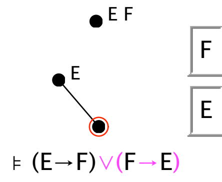


Fig. 11. An isolated world

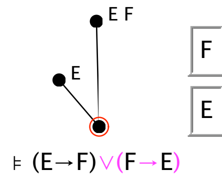


Fig. 12. A different connection

4 Exploring diagrams

The definition of $A \rightarrow B$ – where A is forced, B must be forced as well – is the glory of Kripke's model. It's also quite hard to grasp at first. I give some assistance to novices by allowing them to experiment with different situations, and more by allowing them to explore a situation – counter-example or not – to see where particular subformulae are forced.
Figure 13, for example, is the result of clicking on one of the worlds of figure 10. The colouring of the assertion is now not as it was when the base world was selected. Occurrences of E are now violet, but F is still black (atomic assertions are forced if they are present, not forced if absent). The black arrow shows that $E \rightarrow F$ is not forced, and the colouring of the basic assertions shows that that's because E is forced and F isn't; similarly, it shows why $F \rightarrow E$ is forced. The disjunction, which is the overall assertion, is still forced because one of its components is forced.
That situation shows the novice why $E \rightarrow F$ isn't forced in the base world – it must hold everywhere above the base world, not just in one place, and the intermediate world contradicts it. Clicking on the top world makes the same point more strongly, producing the classical single-world evaluation of figure 14. Now everything is forced: both the atomic assertions, both the implications and the disjunction. Implication, in this semantics, is a promise rather than a static property, and a promise broken in one world is broken in all the worlds below it. Although $E \rightarrow F$ holds in the top world, and it would hold in the root world if it didn't have descendants, it doesn't hold in the intermediate world and that's enough to say that it doesn't hold in any world which has that one as a descendant.

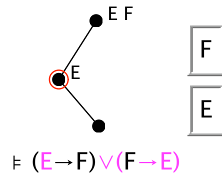


Fig. 13. Another evaluation

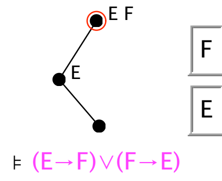


Fig. 14. A classical evaluation

Dragging and dropping is extensively supported in Jape's treatment of constructive disproof. You can drag worlds onto worlds or lines, and lines onto worlds. You can drag anything – worlds, labels, lines – into the swing bin (it's the most straightforward mechanism for deletion, vastly easier to use than a "select; delete" technique). Using drag and drop, plus an occasional focus-click, for every action in the proof pane is a worthwhile simplification, a reasonably quiet interface.

5 Working with quantifications

Forcing semantics of quantification relies on named individuals inhabiting a universe. Adding individuals to the logic, via the pseudo-assertion actual <*name*>, makes it possible to give a much more coherent account of Jape's proof capabilities. Using individuals in disproof requires a little care, and Jape is not yet completely developed in this direction.
Consider, for example, the assertion

$$\mathrm{actual}\, i,\ \forall x.\, (R(x) \vee \neg R(x))\,,\ \neg\forall y.\neg R(y)\ \vDash\ \exists z.R(z)$$

It looks unexceptionable: in a non-empty universe (there is at least the individual i), where R is a decidable property, not false everywhere, there must be some individual with property R. But it's one of the entries in Jape's "Classical conjectures" panel, so even the novice can guess that it must have a constructive disproof as well as a classical proof.
The proof, shown in figure 15, is easy once you realise that the decidability hypothesis was inserted as a sop to the constructivists, and even the non-emptiness of the universe is nothing to do with anything. But the constructive attempt, though it uses all the premises and explores more avenues, gets stuck as in figure 16 because it can't make the same first move.
Never mind! The stuck proof tells us what to do: make a world with an individual i (line 1), another with $i1$ (line 7), don't have $R(i)$ (line 6), but do have $R(i1)$ (line 8). Jape's starting point for a disproof, shown in figure 17, colours actual i black, because it doesn't occur at the root world, underlines the second premise, which is forced trivially because there are no individuals in the diagram, and doesn't underline the third for similar reasons. The conclusion then isn't forced because there is no individual to stand witness. The tiles on the right allow us to make worlds involving the only individual (i) and the only predicate (R) in the assertion. They won't be enough, but we can make a start.
Recall that to produce a counter-example we must make a situation in which all the premises are forced and the conclusion is not. So the first step must be to force actual i in the root world (figure 18). Observe that actual i is now forced, and both $\forall$s are too, because $\neg R(i)$ is now forced. The $\exists$ is still black. And notice that the formulae inside the quantifications are grey: a new kind of colouring.[1] Greyness signifies that a formula isn't strictly relevant to the status of the assertion, and that's

[1] For those of you reading in black and white, as they almost used to say on UK television, the grey bits look just like the violet bits. Sorry!

1:	actual i	premise
2:	∀x.(R(x)∨¬R(x))	premise
3:	¬∀y.¬R(y)	premise
4:	¬∃z.R(z)	assumption
5:	actual i1	assumption
6:	R(i1)	assumption
7:	∃z.R(z)	∃ intro 6,5
8:	⊥	¬ elim 7,4
9:	¬R(i1)	¬ intro 6–8
10:	∀y.¬R(y)	∀ intro 5–9
11:	⊥	¬ elim 10,3
12:	∃z.R(z)	contra (classical) 4–11

Fig. 15. A classical success

1:	actual i	premise
2:	∀x.(R(x)∨¬R(x))	premise
3:	¬∀y.¬R(y)	premise
4:	R(i)∨¬R(i)	∀ elim 2,1
5:	R(i)	assumption
6:	¬R(i)	assumption
7:	actual i1	assumption
8:	R(i1)	assumption
	. . .	
9:	⊥	
10:	¬R(i1)	¬ intro 8–9
11:	∀y.¬R(y)	∀ intro 7–10
12:	⊥	¬ elim 11,3
13:	R(i)	contra (constructive) 12
14:	R(i)	∨ elim 4,5–5,6–13
15:	∃z.R(z)	∃ intro 14,1

Fig. 16. A constructive failure

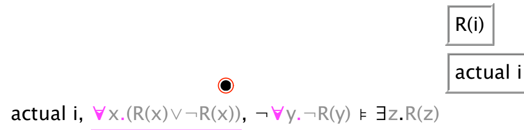


Fig. 17. The start of a constructive disproof

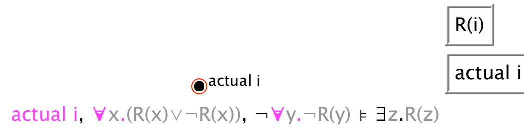


Fig. 18. An individual added

true in this case, because $R(x)$ and $R(y)$ aren't interesting in a universe containing an individual i: only $R(i)$ matters. I shall return to the issue of evaluation and colouring of quantified formulae below.

We can press on. The stuck proof suggests (because of its box structure) that we should make a new world containing an individual $i1$. It's easy to make a new world by option-dragging the root world, but we don't have a tile with actual $i1$. Jape lets us add individuals to the model by double-clicking an 'actual' tile. This solves our immediate problem, and we can drag and drop the result onto the upper world (figure 19).

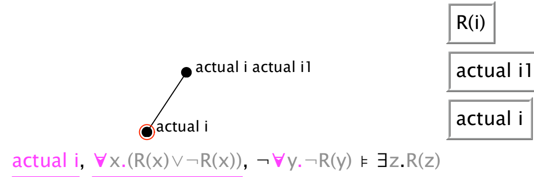


Fig. 19. A new world and a new individual

We still don't have $R(i1)$ to drop onto the model, but double-clicking a predicate tile gets you a novel instance built up from the available individuals (if there is more than one possibility, a choice dialogue lets you pick the one you want). That new instance can be dropped onto the diagram and the disproof is complete (figure 20) – all premises forced (shown by underlining) and the conclusion unforced.

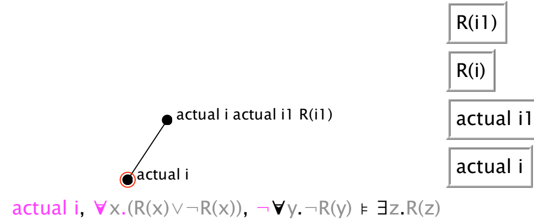


Fig. 20. A completed disproof

This is all very well: the stuck proof shows the expert how to proceed, and the interface supports all the necessary moves. But the display doesn't help the novice to understand *why* it's colouring the assertion as it does. Most of the syntax colouring is grey, telling us that the literal subformulae aren't used, and that there is more behind the scenes to be explained. Exploring the worlds doesn't help much at all, as figure 21 shows.

At this point I have to say that, as a designer, I'm stumped. I'd like to design an interaction that allows a novice to double-click (say) on $\exists z.R(z)$ and find out why it isn't forced at the base world (there's only

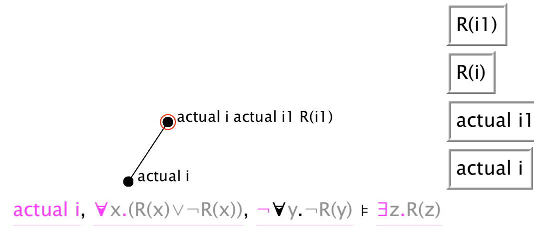


Fig. 21. A lot of colour and little illumination

one individual available and $R(i)$ isn't forced at the base world) or why conversely it is forced in the upper world (there we can call on $i1$ and $R(i1)$). That might be supported, in this trivial example, by a little drop-down menu, and the results might be helpful. But in a more complex example using nested quantifiers we would want to explore further, and need to explore what we see in a way that drop-down menus in the assertion don't seem to support.

6 Deficiencies

Explanation is partly supported in the propositional case, hardly supported at all in the predicate case, as discussed above. I suspect this is a considerable stumbling block for the average novice.
Layout of the labels decorating a world is primitive: they are shown as a single line of text extending right from the world. In a large diagram those text lines confusingly overlap other objects.
When drawing complex structures a single-history undo is sometimes irksome. It would be worth experimenting with an object-focussed undo, though I'm conscious of the need to preserve a quiet interface.

7 Evaluation

This mechanism hasn't been independently evaluated. It's been given to two successive first-year classes of undergraduates to use, in the first year under my supervision. My informal assessment is that students find Jape's proof mechanisms, especially since they were subjected to thoughtful evaluation in [2], relatively easy to use, and formal proof correspondingly relatively easy to learn. They find disproof harder to understand and in practice they struggle with the disproof interface. (Then most of them then find Hoare logic, which was taught without Jape support in the final section of the course, quite impenetrable, but that's another story!)
On the other hand many of them do learn about models and disproof using Jape, and all appear to be grateful for the unbiased verdict of a calculator on their attempts to answer exercises. The interaction with the disproof calculator is much more like what many of them would hope to

have with a proof calculator: they can put in a copy of what they have written on paper, and receive a judgement, provided that their pencil and paper version is not too outlandish.
Clearly it would need a dispassionate evaluation by a trained educationalist to tease out the interface difficulties from the conceptual misunderstandings, so that interactive disproof in Jape can perhaps become as smoothly successful as its proof mechanisms.

Acknowledgements

Bernard Sufrin and I designed and implemented Jape together for several years. His architectural insight, and crucial early design decisions, have been essential underpinning for everything I have managed to achieve since he moved on to other things. His continued advice helps me implement along the narrow path.
David Pym persuaded me to take Kripke semantics seriously as an educational target. The hundred and eighty students who suffered the first time I tried to teach it without mechanical support provided the evidence which underpinned the case for a disproof calculator.
Paul Taylor, Jules Bean, Mike Samuels and Graham White (who invented the 'scrabble tile' layout) advised me and supported me whilst I built the disproof parts of Jape. Paul's ideas on explanation were particularly influential.
Even more than usual, mistakes in Jape are all my own fault.

References

1. The jape web site. http://www.jape.org.uk. Latest versions of Jape for various platforms, including the encoding of natural deduction discussed in this paper.
2. J. C. Aczel, P. Fung, R. Bornat, M. Oliver, T. OShea, and B. Sufrin. Using computers to learn logic: undergraduates experiences. In G. Cumming, T. Okamoto, and L. Gomez, editors, *Advanced Research in Computers and Communications in Education: Proceedings of the 7th International Conference on Computers in Education*, Amsterdam, 1999. IOS Press.
3. R. Bornat and B. Sufrin. Jape: A calculator for animating proof-on-paper. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 412–415, Berlin, July 13–17 1997. Springer.
4. Richard Bornat. Rendering tree proofs in box style. submitted to UITP 03, 2003.
5. Richard Bornat and Bernard Sufrin. Animating formal proof at the surface: The Jape proof calculator. *The Computer Journal*, 42(3):177–192, 1999.
6. Richard Bornat and Bernard Sufrin. A minimal graphical user interface for the jape proof calculator. *Formal Aspects of Computing*, 11(3):244–271, 1999.

7. M. Dummett. *Elements of Intuitionism*. Oxford Logic Guides. Clarendon Press, Oxford, 1977.
8. S. A. Kripke. Semantical analysis of intuitionistic logic. In J. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, Amsterdam, 1965.

Taclets and the KeY Prover

Extended Abstract

Martin Giese

Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden
giese@cs.chalmers.se

Abstract. We give a short overview of the KeY prover, which is the proof system belonging to the KeY tool [1], from a user interface perspective. In particular, we explain the concept of *taclets* which is the basic building block for proofs in the KeY prover.

1 Introduction

The goal of the ongoing KeY Project [1] is to make the application of formal methods possible and effective in a real-world software development setting. One of the main products of the KeY Project is the KeY Tool, which allows the specification and verification of Java Card [8] programs. The KeY Prover is an integrated interactive and automated theorem prover that is used in the KeY tool to reason about programs and specifications.

The logic employed by the KeY prover is a dynamic logic (DL) for Java Card [2]. This can be viewed as a kind of first order multi-modal logic, where modal operators are indexed by programs. A diamond formula $\langle\pi\rangle\,\phi$ means that there is a terminating execution of the program π after which ϕ holds, a box formula $[\pi]\,\phi$ means that ϕ holds after every terminating execution.

Most of the proof rules available in the KeY system symbolically execute programs in DL formulae. There are also some induction rules to reason about loops and recursion. The "core" of the calculus is however first-order, in the sense that there is no quantification over functions or sets, no lambda abstraction, etc. The prover uses a sequent-style calculus, which is augmented with *meta variables* to allow the delayed choice of quantifier instantiations, similarly to the free variables used in first order tableau calculi.

As program verification cannot be done fully automatically for realistic programs, it was important to make the interactive user interface of the KeY prover intuitive and powerful.

The KeY tool can be downloaded free of charge from the KeY project home page at http://i12www.ira.uka.de/~key.

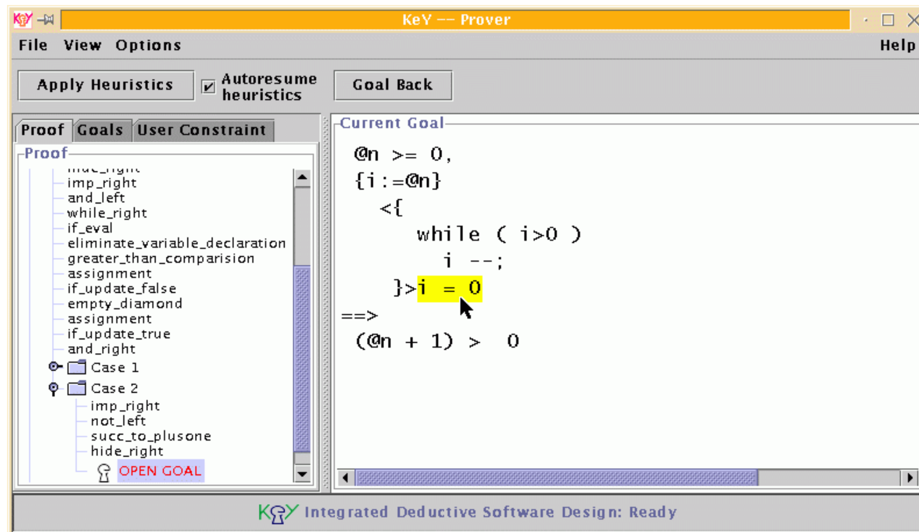


Fig. 1. The main window of the KeY prover

2 The Prover Window

In Fig. 1, the main window of the KeY prover is shown. In the left part of the window, the whole proof tree is displayed, showing the applied rules and the case distinctions, which correspond to splits in the proof tree. Using the "tabs", one can also choose to display only a list of open goals, or the *user constraint*. The user constraint allows the user to control the instantiation of meta variables. The proof steps displayed in the proof view have pop-up menus which allow the user e.g. to cut off parts of the proof at a certain point.

The right part of the window displays the sequent that is currently being worked on. A formatting engine in the style of Oppen's pretty-printer [7] is used to print sequents with a structured layout. As is visible in the figure, the formula, sub-formula, term, etc. that is currently under the mouse pointer is highlighted. Highlighting, in conjunction with layout, helps the user in understanding the structure of a complex formula.

Clicking on an operator in a formula displays a pop-up menu giving a choice of rule applications possible for that sub-formula, see Fig. 2. In the KeY prover, the rules from which proofs are built are combined with the information of how the user should interact with these rules, to form entities called *taclets*, see next section.

We display the proof tree and the current sequent together in one window, using a split pane, for the following reason. It makes perfect sense to work on several proofs at a time. For instance, during construction of a new proof, one might want to consult an older one for reference. It is also conceivable, though not yet implemented, to cut and paste parts of proofs. In such a setting, having

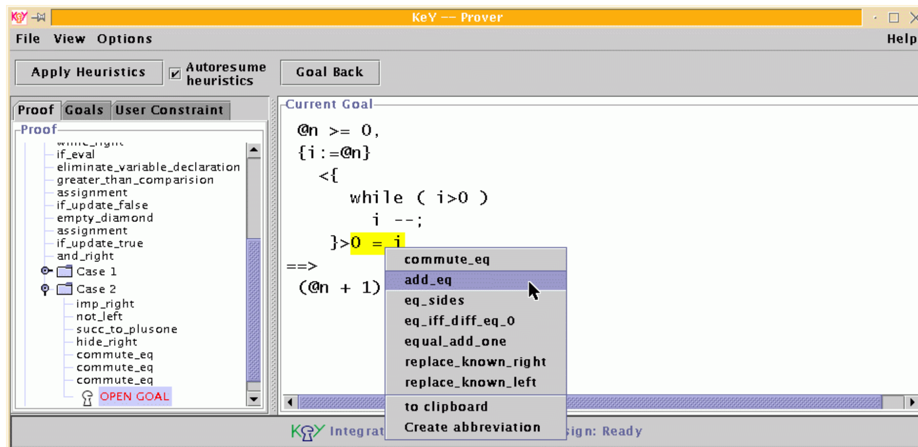


Fig. 2. Choosing a taclet to apply

separate windows for proof trees and sequents would make it hard for the user to see which belongs to which.

3 Taclets

Most existing interactive theorem provers are "tactical theorem provers". The tactics for which these systems are named are programs which act on the proof tree, mostly by many applications of primitive rules, of which there is a small, fixed set. The user constructs the proof by selecting the tactics to run. Writing a new tactic for a certain purpose, e.g. to support a new data type theory, requires expert knowledge of the prover.

In the KeY prover, both tactics and primitive rules are replaced by the *taclet* concept.[1] A taclet combines the logical content of a sequent rule with pragmatic information that indicates how and when it should be used. In contrast to the usual fixed set of primitive rules, taclets can easily be added to the system. They are formulated as simple pattern matching and replacement schemas. For instance, a very simple taclet might read as follows:

find (b −> c ==>) if (b ==>) replacewith(c ==>) heuristics(simplify)

This means that an implication b −> c on the left side of a sequent may be replaced by c, if the formula b also appears on the left side of that sequent.

Apart from this "logical" content, the keyword find indicates that the taclet will be attached to the implication and not to the formula b for interactive

[1] Taclets have been introduced under the name of *schematic theory specific rules (STSR)* by Habermalz [6]. The concept of interactive theorem proving through direct manipulation of formulae was inspired by the theorem prover InterACT [4].

selection, i.e. it will appear in the pop-up menu when the implication is clicked on.

Taclets can be part of *heuristics*. The clause heuristics(simplify) indicates that this rule should be part of the heuristic named simplify, meaning that it should be applied automatically whenever possible if that heuristic is activated. The user can interactively change which heuristics should be active at a certain time.

While taclets can be more complex than the typically minimalistic primitive rules of tactical theorem provers, they do not constitute a tactical programming language. There are no conditional statements, no procedure calls and no loop constructs. This makes taclets easier to understand and easier to formulate than tactics. In conjunction with an appropriate mechanism for heuristic application, they are nevertheless powerful enough to permit comfortable interactive theorem proving [6]. For the automated execution of heuristics, the idea is that any possible taclet application will eventually be executed (fairness), but certain taclets may preferred by attaching priorities to them.

Also note that taclets are rather lightweight entities. It is for instance absolutely possible to introduce dozens of *ad-hoc* taclets to reason about some specific data type in an intuitive way. The set of taclets should and can be designed in such a way that usual human reasoning about some application domain is reflected by the available taclets. An important consequence of attaching taclets to operators is that the taclets for a certain data type will almost all be attached to operators of the according type. For instance, taclets for reasoning about numbers are attached to operators like + or >=, etc. This means that when the user clicks on a specific operator, only those taclets will be visible that are relevant for that operator in that context. This significantly reduces the burden on the user that is usually associated with a large set of rules.

In principle, nothing prevents the formulation of a taclet that represents an unsound proof step. It is possible however, to generate a first-order proof obligation from a taclet, at least for taclets not involving DL. If that formula can be proved using a restricted set of "primitive" taclets, then the new taclet is guaranteed to be a correct derived rule.

No provision is currently made in the user interface for the *construction* of taclets. They are given in the textual form shown above and read into the system by a parser. In future versions, a possibility to define taclets within the user interface might be added to the system.

4 Implementation

The KeY prover is implemented in the Java programming language [5], using the Swing [9] GUI library. The coordination between the displayed proof tree, the current sequent, etc. and the underlying logical data structures follows the *Model, View, Controller* architecture, making intensive use of the *Listener* design pattern (see [3]). While this is not the fastest conceivable technique, it has helped to provide a good modularization of the system.

Highlighting and generation of position-dependent pop-up menus depends on having a fast mechanism to find the term position corresponding to a certain character in the displayed sequent. This is achieved using *position tables*, which record the start and end of nested formulae and terms in every sub-formula/term of the sequent. Position tables are built by the pretty-printer during layout, at a low additional cost, and they are very efficient. There is no perceivable delay due to highlighting when the mouse is moved over the sequent.

For a pleasant user experience, it is also important that the available taclets at a certain position are displayed with minimal delay when the user clicks somewhere. This is achieved using a number of indexing data structures. For every open goal, a *taclet application index* is kept, that stores all taclet applications possible in a sequent at any position. It is organized in such a way that quick access to the applicable taclets is possible based on the position in sequent. Only taclet applications that are actually possible are stored. Regard for instance the taclet given in the previous section, which requires the presence of b in the antecedent. If that formula is not present, a corresponding taclet application will not be put into the taclet application index, and thus will not be displayed to the user. In the current implementation, the taclet application index is simply recalculated before each interaction, but it would be possible to cache most taclet applications between taclet applications, as most of the sequents remain unchanged.

In order to calculate the taclet application index efficiently, a *taclet index* is used. This contains the set of all available taclets, and provides an operation to determine a set of candidates that might be applicable, given some formula and its position in a sequent. The idea is to go through all sub-formulae of a newly introduced formula in a sequent and ask the taclet index for a (hopefully small) set of potentially applicable taclets. It is then checked whether all conditions of the taclet actually are satisfied, and if so, a corresponding taclet application is put into the taclet application index. What indexing mechanism is sensible for the taclet index is of course dependent on the set of taclets in use. Many of the taclets currently used in the KeY prover serve the symbolic execution of programs. We use a hash table indexed by the top operator, and in case of program modalities, by the type of the first executable statement in the program in question. This gives very acceptable performance for interactive use: the time required to apply a rule, to build the new taclet application index and to layout and display the new sequent lies mostly below half a second. The standard set of taclets usually worked with comprises several hundred taclets for propositional and predicate logic, integers, sets and above all for Java Card. When taclets are applied automatically by the "heuristics", performance ranges between 10 rule applications per second for the more complicated symbolic execution taclets to several hundred per second for simple propositional logic.

The performance might become unacceptable in the future, due for instance to an enlarged taclet base. In that case, our course will be to progressively optimize the indexing data structures. In fact, this has already been done twice in the past: originally there was no taclet index at all. As the number of predicate

logic rules grew, hashing on the top function symbol was introduced. Finally, with the addition of DL rules, indexing on program statements became necessary.

Another conceivable future optimization is to compile taclets: As taclets have a quite operational semantics, it would be possible to produce Java byte code for the actions of a taclet, instead of the current interpretive approach. It is not clear weather this will become necessary, as the system performs quite satisfactorily so far.

5 Conclusion

We have briefly described the KeY prover from a user interface perspective. In particular, we have introduced the concept of *taclets*, which consist of the logical content of a sequent rule, paired with pragmatic information on how and when to apply it. We have also given a short overview of some of the non-trivial implementation issues involved.

Acknowledgments

The author is indebted to Richard Bubel for providing some of the technical details and to Wolfgang Ahrendt for helpful comments on a draft version of this paper.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, February 2003.
2. Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In Isabelle Attali and Thomas P. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading/MA, 1995.
4. R. Geisler, M. Klar, and F. Cornelius. *InterACT*: An interactive theorem prover for algebraic specifications. In *Proc. AMAST'96, 5th International Conference on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 563–566. Springer, July 1996.
5. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1997.
6. Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000. `http://i12www.ira.uka.de/~key/doc/2000/stsr.ps.gz`.
7. Derek C. Oppen. Pretty-printing. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

8. Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997.
`ftp://ftp.javasoft.com/docs/javacard/JC20-Language.pdf`.
9. Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison Wesley, 1999.

Interactive Proof Construction at the Task Level

Malte Hübner, Christoph Benzmüller, Serge Autexier and Andreas Meier

Fachbereich Informatik, Universität des Saarlandes
D-66041 Saarbrücken, Germany
{huebner|chris|serge|ameier}@ags.uni-sb.de

Abstract. Interactive theorem proving systems for mathematics require user interfaces which can present proof states in a human understandable way. Often the underlying calculi of interactive theorem proving systems are problematic for comprehensible presentations since they are not optimally suited for practical, human oriented reasoning in mathematical domains. The recently developed CORE theorem proving framework [Aut03] is an improvement of traditional calculi and facilitates flexible reasoning at the assertion level. We make use of COREs reasoning power and develop a communication layer on top of it, called the *task layer*. For this layer we define a set of manipulation rules that are implemented via COREs calculus rules. We thereby obtain a human oriented interaction layer that improves and combines ideas underlying the window inference technique [RS93], the proof by pointing approach [BKT94], and the focus windows of [PB02].

1 Introduction

In interactive theorem proving (ITP) it is important that information about a proof state is presented in an intuitive way in order to help the user in the selection of a next proof step. Ideally, the knowledge (assertions) available to prove a focused goal (i.e. its logical *context*) is presented in a suggestive manner supporting a uniform handling of assertion selection and application. Consider for example a situation where a goal formula B has to be derived from some formulas $Ax_1, \ldots, Ax_n$ (we say that $Ax_1, \ldots, Ax_n$ are in the context of B). If we further assume that the formula $(\forall x.Q[x]) \Rightarrow (B \wedge C)$ is amongst the Ax_i then we could present this situation in the form shown in Figure 1(a). Using this representation the formulas in square brackets represent all the information that is available to infer the goal B. In the situation above it seems promising to apply the assertion $(\forall x.Q[x]) \Rightarrow (B \wedge C)$ to transform B into $\forall x.Q[x]$ as depicted in Figure 1(b).

In most current ITPs such a style of presentation and reasoning is difficult. The reason for this is that these ITPs are typically based on calculi that have not been developed for practical reasoning but for proof-theoretic considerations. These calculi often represent a proof state as sets of sequents or clauses and sometimes even require formulas to be in a certain normal form which makes a presentation like the above difficult. Furthermore, assertions can often not be

$$a) \begin{bmatrix} Ax_1 \\ \vdots \\ Q[x^\gamma] \Rightarrow (B \wedge C) \\ \cdot \\ Ax_n \\ A \end{bmatrix} \qquad b) \begin{bmatrix} Ax_1 \\ \vdots \\ Q[x^\gamma] \Rightarrow (B \wedge C) \\ \cdot \\ Ax_n \\ A \end{bmatrix}$$

$$B \qquad \qquad \forall x.Q[x]$$

Fig. 1. Intuitive presentation of proof tasks: goals appear together with the assertions available in the context.

applied directly to a subgoal and have to be explicitly decomposed through the application of calculus rules before they can be used in a proof. The need for explicit decomposition is one reason why natural deduction style calculi and sequent style calculi are far less intuitive and human oriented as we actually wish.

Autexier [Aut01,Aut03] recently developed the CORE theorem proving system as a means for the integration of multiple proof paradigms into one framework. The system supports a proof style that is based on *contextual rewriting* and it furthermore exploits ideas of the *window inference technique* [RS93]. The novelty of the system lies in the fact that it does not make use of a fixed set of calculus rules that are defined over the syntactic structure of formulas; rather, it allows the user to freely focus the reasoning process on subformulas of the overall goal formula while the system internally updates the logical context of these formulas while avoiding explicit decomposition. The context of a subformula is then made available to the user in form of transformation rules, so called *replacement rules*, which can be used to directly transform the subformula in the current focus. One of the advantages of this reasoning style is that it supports the application of assertions in a more intuitive way, while formula decomposition is treated implicitly. In fact, the stepwise unwrapping of subgoals and assertions is replaced by flexible focus relocations and replacement rule applications. However, although CORE is a potentially well suited basis for ITP it is not yet free of problems. A major drawback is that, until now, it is only possible to display the context of a formula as a usually long and unstructured list of replacement rules.

In this paper we develop a uniform communication layer — called *task structure* — on top of the CORE reasoning framework. This communication layer is intended to serve as the exclusive interface between CORE and the user and between CORE and automated proof procedures, such as the proof planner MULTI [Mei03] of the ΩMEGA mathematical assistant system [SBF+03]. Task structures allow to display the formulas in the context of a formula in the style used in Figure 1 and also provide a uniform framework for application of these formulas. The overall idea is that reasoning on the task structure reflects flexible reasoning with assertions while all logic level aspects (including decomposition) are real-

ized via CORE and thus hidden from the user. We thereby gain a system that improves and combines the ideas of the window inference technique [RS93], the proof by pointing approach [BKT94], and the focus windows of [PB02]. By mapping the proof steps performed at the task level into sequences of CORE calculus rule applications we can furthermore guarantee the soundness of the steps taken at task level.

The remainder of the paper is outlined as follows. In Section 2 we introduce the CORE system before we describe the task structure in Section 3. Section 4 demonstrates the benefits of our approach at hand of a small example.

2 The CORE System

CORE supports flexible contextual rewriting: When focusing on a subformula F' of a goal F for manipulation the CORE system determines the logical context of F' and makes it available to the user as a set of replacement rules R. The replacement rules in R can be directly employed for manipulation of F'.

This supports a natural way of reasoning which treats formula decomposition implicitly and which resembles Huang's [Hua94] reasoning on the assertion level. We illustrate this at hand of the following theorem:

$$(M \wedge N) \wedge (M \wedge N \Rightarrow P) \Rightarrow P \tag{1}$$

In CORE we can use the implication $M \wedge N \Rightarrow P$ in the context of P to replace P by $M \wedge N$. This is realized with the help of the replacement rule $P \rightarrow< M \wedge N >$ which is generated from the respective implication. Application of this replacement rule to P thus yields

$$(M \wedge N) \wedge (M \wedge N \Rightarrow P) \Rightarrow M \wedge N \tag{2}$$

The newly introduced $M \wedge N$ can now be replaced by *true* through application of the replacement rule $(M \wedge N) \rightarrow< \mathit{true} >$ which we obtain from the subformula $(M \wedge N)$ on the left hand side of the implication. Thus, we have

$$(M \wedge N) \wedge (M \wedge N \Rightarrow P) \Rightarrow \mathit{true} \tag{3}$$

which CORE simplifies to *true*.

This small example illustrates COREs key characteristics: Reasoning at the assertion level is made possible through the generation of replacement rules from the assertions in the context of a subformula. Furthermore, the proof problem is always represented and maintained in its entirety instead of being decomposed into smaller pieces as in sequent- and natural deduction style calculi.

2.1 Proof Theoretic Annotations

CORE is based on the notions of *signed formulas* and *indexed formula trees* (IFTs) which go back to Smullyan [Smu68] and Wallen [Wal90]. We only briefly

α	α_1	α_2
$(A \wedge B)^-$	A^-	B^-
$(A \vee B)^+$	A^+	B^+
$(A \Rightarrow B)^+$	A^-	B^+
$(\neg A)^+$	A^-	
$(\neg A)^-$	A^+	

β	β_1	β_2
$(A \wedge B)^+$	A^+	B^+
$(A \vee B)^-$	A^-	B^-
$(A \Rightarrow B)^-$	A^+	B^-

Fig. 2. Uniform Notation (Propositional Types)

δ	δ_0
$(\forall x.F[x])^+$	$F[x/c]^+$
$(\exists x.F[x])^-$	$F[x/c]^-$

γ	γ_0
$(\forall x.F[x])^-$	$F[x/c]^-$
$(\exists x.F[x])^+$	$F[x/c]^+$

Fig. 3. Uniform Notation (Quantifiers); respective variable conditions are introduced and maintained by CORE for these cases

sketch them here and refer to [Aut03] for a more detailed introduction. CORE supports a variety of different logics, including higher-order and modal logics, however, for sake of simplicity, we only consider a classical first-order system in this paper.

Definition 1 *(Signed Formulas) A* signed formula *is a pair (A, p), where A is a formula and $p \in \{+, -, 0\}$ the* polarity *of A. We usually write A^p instead of (A, p).*

Signed formulas are assigned a uniform type to distinguish between disjunctive formulas (type α), conjunctive formulas (type β), universal formulas (type γ) and existential formulas (type δ). The tables in Figures 2-4 define the types of signed formulas and also determine how type information is propagated to the major subformulas of a signed formula.

In the remainder we denote by $\alpha(\alpha_1^{p_1}, \alpha_2^{p_2})^p$ a signed formula F of type α and polarity p. The $\alpha_i^{p_i}$ are the major subformulas of F with polarities p_i. Formulas of type β, γ and δ are denoted in a similar way. To indicate the type of a formula we frequently attach the type information to the topmost connective of a formula, e.g. $(A \wedge^\beta B)^+$. We furthermore abbreviate $\alpha(F_1, \alpha(F_2, \ldots, \alpha(F_{n-1}, F_n)))$ as $\alpha(F_1, \ldots, F_n)$ and use a respective notation for β.

Intuitively, the information encoded in annotated signed formulas describes the "behavior" of these formulas in sequent calculus derivations. The polarity $(+/-)$ of a signed formula is just another representation of the succedent/antecedent distinction used by Gentzen [Gen35] in the sequent calculus. Wallen [Wal90] interpretation is: Instead of distinguishing between formulas in the antecedent Γ and succedent Δ by using the sequent symbol $\vdash$ (i.e. $\Gamma \vdash \Delta$), he simply annotates all formulas that would occur in the antecedent Γ of a sequent (after application of the appropriate sequent calculus rules) with a negative polarity $(-)$, and formulas that would occur in the succedent of a sequent with a positive $(+)$ polarity. The uniform type of a formula describes whether the subformulas into

ϵ	ϵ_1	ϵ_2
$(A \Leftrightarrow B)^-$	A^0	B^0
$(s = t)^-$	s^0	t^0

ζ	ζ_1	ζ_2
$(A \Leftrightarrow B)^+$	A^0	B^0
$(s = t)^+$	s^0	t^0

Fig. 4. Uniform Notation (Equivalences and Equations); the major subformulas of equivalences and equations are of polarity 0 (the undefined polarity)

which the formula would be decomposed after application of the corresponding sequent calculus rule occur together in the same sequent (i.e. in the same branch of the proof) or will be parts of different sequents (i.e. different proof branches). In the former case the formula is assigned the uniform type α while in the latter case it has uniform type β.

Furthermore, the types δ and γ are assigned to formulas with a quantifier as topmost logical connective (i.e. formulas of the form $Qx.F[x]$ with $Q \in \{\forall, \exists\}$). Type δ is assigned to formulas which topmost quantifier binds a variable with an *Eigenvariable condition* and γ refers to freely instantiable variables. For more details on the respective variable conditions introduced and maintained by CORE for these cases we refer to [Aut03].

Negative equations and equivalences are of type ϵ and their constituents have undefined polarity. ζ and ϵ formulas are displayed in Figure 4.

Signed formulas F can be represented as trees where each node is labeled with a signed subformula of F. For each node n with label N in such a tree we require that the child nodes of n are labeled with the major subformulas of N.[1] This leads to the notion of an *indexed formula tree* (IFT). The IFT for formula (1) is given in Figure 5. For a formal definition of an IFT we refer to [Aut03] and [Wal90].

We say that two nodes of an IFT are β-related (respectively α-related) to each other if they have a common father (the first common predecessor) that is labeled with a signed formula of type β (respectively α).

We distinguish between dependent and independent nodes in an IFT.

Definition 2 *(Dependent Occurrence) Let a be a node in an* IFT *R. We say that a is* dependent *in tree R if there is a node b in R that is β–related to a. Otherwise we call a an* independent occurrence *in R.*

It it useful to introduce a partial ordering $\prec$ on the set of nodes of IFTs: $n_1 \prec n_2$ if and only if n_1 is an ancestor of n_2.

Indexed formula trees are used in CORE to represent the quantifier dependencies of a proof state in the following way: When we load a goal formula G together with axioms $Ax_1, \ldots, Ax_n$ from which we want to derive G the system creates an IFT for the signed formula $(Ax_1 \wedge \ldots \wedge Ax_n \Rightarrow^\alpha G)^+$. The system also creates a *free variable indexed formula tree* (FVIFT) for the same formula. FVIFT and IFT together represent a CORE proof state where proof search manipulates the FVIFT while the IFT is used to maintain the dependencies between quantifiers. The FVIFT can be easily obtained from an IFT by simply removing

[1] We define F and G to be the major subformulas of $F \wedge G, F \vee G, F \Rightarrow G, \neg F$. Any $Q[x/t]$ is a major subformula of the formulas $\forall x.Q[x]$ and $\exists x.Q[x]$.

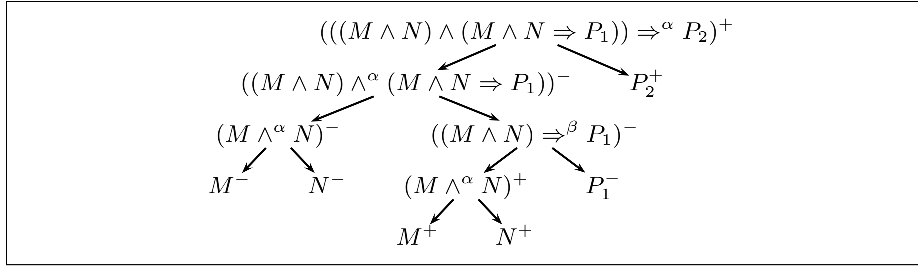


Fig. 5. Indexed formula tree for $(((M \wedge N) \wedge (M \wedge N \Rightarrow P)) \Rightarrow^{\alpha} P)^+$. Subscripts of literals are added for later reference.

all nodes of type γ and δ. Note that when we remove a node of type γ we must α-relate its children. For our propositional logic example formula (1) the Fvift is identical to its Ift displayed in Figure 5.

2.2 Logical Context and Replacement Rules

We have already referred to the concept of a context of a subformula inside a larger formula. This concept is essential for the understanding of CORE. Making use of annotations of signed formulas we are now able to give a more precise definition of the *logical context* of a formula: two formulas F and G belong to the same context if they are α-related.

Intuitively, all formulas that would occur together in a sequent calculus derivation of a goal G belong to the same context. Note that the information available in Ifts and Fvifts is sufficient to statically determine the logical context of any given subformula.

In CORE we can use the formulas available in the context of a subformula directly as replacement rules in a proof. We formally define the notion of replacement rules as follows:

Definition 3 *(Replacement Rules) Let a be a node with polarity p in some* Fvift T*. Then $i \rightarrow < v_1, \dots v_n >$ is a* replacement rule *for a, if*

1. *i is α-related to a by the first common ancestor c, and*
2. *(a) $\{v_1, \dots, v_n\}$ contains exactly those occurrences that are β-related to i and occur below c (i.e. $c \prec v_i$), or*
   (b) v_1 and i are left- and right-hand side of a negative equation or equivalence and $\{v_2, \dots, v_n\}$ are all occurrences that are β-related to v_1 and i and occur below c.

A positive (negative) occurrence M^+ (M^-) without any β-related nodes generates $M^- \rightarrow true^+$ ($M^+ \rightarrow false^-$) as a replacement rule.

CORE provides a set of 12 "calculus rules" which can be used to manipulate Ifts. The rule that is most important for us is the one that applies a replacement rule to a subformula. We do not formally introduce the complete set of calculus

rules here and instead refer to [Aut03]. For the realization of the task structure another aspect of CORE is more important which is known as *window inference*.

2.3 Window Inference

On top of the calculus rules described above, CORE supports reasoning with a so called *window inference technique* which is based on ideas of [RS93]. In this section we describe this communication layer which makes it possible to focus the reasoning process to subformulas of the overall goal. This can be done by placing a *window* (focus) on a subtree of the FVIFT. As a result, the surroundings of this window are hidden from the user. However, this operation does not alter the proof state but only restricts the view on it to a particular subtree. The context of the active window is then made available to the user as a list of replacement rules.

The window inference mechanism of CORE consists of rules to focus and unfocus certain subformulas as well as window versions of each of the CORE calculus rules. It is worth pointing out that these window inference rules do not extend the reasoning capabilities of CORE. Rather, the window versions of the calculus rules are internally realized purely based upon the CORE calculus rules and the rules for opening and closing windows.

Opening and Closing Windows The two basic window rules are those for opening and closing windows on subformulas (subtrees). By applying these operations to a FVIFT we impose a *window structure* on this tree. As a window structure for a FVIFT R we understand a partial function f_R from an enumerable set W of unique window identifiers into the set of subtrees of R (i.e. their roots). Intuitively, when opening a new window w on a subtree r in R then we add the tuple (w, r) to f_R. Closing a window is exactly the inverse operation which removes a tuple from f_R. We will depict a window structure f_R on a FVIFT R by drawing boxes around subtrees r for which a window w exists (i.e. if $f_R(w) = r$). Figure 6 (left) shows the FVIFT for $P(x) \Rightarrow (Q(x) \wedge R(x))^+$ after opening windows on $P(x) \Rightarrow (Q(x) \wedge R(x))^+, P(x)^-$, and $R(x)^+$ respectively. Figure 6 (right) shows the same tree with a new window structure where the window on $R(x)$ is closed.

Windows w that correspond to subtrees $f(w)$ that are maximal with respect $\prec$ are called *active windows*. These are the windows that can be manipulated by application of CORE calculus rules.

Definition 4 *(Subwindows) If w is a window in a* FVIFT *R then the set of all* subwindows *of w is defined as*

$$Subwindows(w) = \{w' | f(w) \prec f(w') \text{ and } w' \text{ is a window in } R\}$$

Windows can be closed, provided that they do not contain any subwindows. Note that opening and closing of windows never affects the FVIFT, but rather the "view" the user takes on the problem represented by the tree.

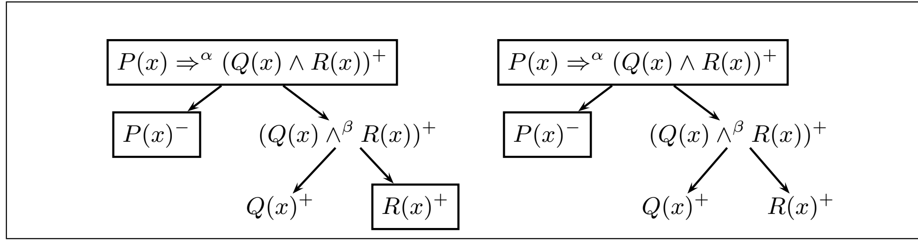


Fig. 6. Sample window structures for the FVIFT for $P(x) \Rightarrow (Q(x) \wedge R(x))^{+}$

2.4 Interactive Theorem Proving with CORE

Within CORE the user currently works with the window inference mechanism. This means that when the user invokes CORE in interactive mode on a goal G with axioms $Ax_1, \ldots, Ax_n$ it assembles an IFT for the formula $(Ax_1 \wedge \ldots \wedge Ax_n \Rightarrow^{\alpha} G)^{+}$ and creates an initial window on G which is presented to the user. The content of this window can then be altered by applying the window versions of COREs calculus rules. Typically this will be an application of a replacement rule. Proof search in CORE is therefore characterized by two major kinds of choices[2]:

1. *Focus choice*: The selection of a subformula in the active window on which the user wants to focus the proof search.
2. *Rule choice*: The selection of a replacement rule.

While COREs inference mechanism is in principle well suited for interactive proof construction, optimal support for focus and rule choice is still challenging. One challenge is related to rule choice since the context of a formula is currently available only as a usually long and unstructured list of replacement rules. This is not what we had in mind when we described the intuitive presentation of a proof in Section 1. To make things even worse, there are already dozens of replacement rules even for rather trivial problems. In particular, the number of replacement rules that can be generated from a subtree of a FVIFT is exponential in the number of nodes in that tree.

This motivates our *task structure* that allows for presenting the context of a subformula in an appropriate way, that is, as a list of assertions. The task structure also supports a uniform application mechanism for the replacement rules associated with these assertions.

3 Tasks – Organizing Proof Search

Informally, a task is an active window together with the assertions in its context. In the following, we formally define tasks and also introduce a set of rules operating on tasks. Thes rules realize the transformations needed for intuitive

[2] CORE also provides a cut-rule which we do not discuss here.

interactive proof construction with CORE (see the previous Section). These rules are in turn implemented via COREs window inference rules. Thus tasks introduce a new interaction layer on top of COREs window inference mechanism.

The concept of tasks was originally developed in the context of automated proof construction with ΩMEGAS [SBF+03] proof planner MULTI [Mei03]. The task structure presented here is an extension and adaptation of this work for the CORE system. An important new idea is to employ tasks as a common and uniform interface for automated and interactive reasoning processes simultaneously.

3.1 A Calculus for Tasks

Intuitively, tasks denote subgoals together with all the formulas that can be used to close this subgoal (all formulas that are α-related to the subgoal). Accordingly a task will simply be defined as a list of windows that all occur in the same context. However, before we make this intuition formal we transfer the notion of dependent occurrences to windows.

Definition 5 *(Conditional Window) Let w be a window on a subformula F in a* FVIFT *R. We say that the window w is* conditional in *R iff the node in R that is labeled with F is dependent in R. Otherwise we call w* unconditional in *R.*

Definition 6 *(Task) Let R be the* FVIFT *of the current proof state. A* task *T is a set of annotated windows $T = \{w_1, \ldots, w_n, g\}$ for R with exactly one* goal window *g and* support windows *$\{w_1, \ldots, w_n\}$. Additionally we require that the following holds if R' is the smallest subtree in R that contains all windows in T:*

1. *the subtrees denoted by the $w_1, \ldots, w_n$ are α-related between each other,*
2. *all support windows of T are unconditional in R'.*

We denote tasks $T = \{w_1, \ldots, w_n, g\}$ with goal window g as $w_1, \ldots, w_n \rhd g$. This notation is a horizontal version of the vertical representation of Figure 1.

Selecting one window as the goal window plays a role when reasoning interactively. The idea is that in an interactive proof only the goal window can be manipulated and therefore we will introduce a *shift*-rule for the explicit exchange of the goal window of a task.

Here we find an important difference to the sequents in the sequent calculus. In the sequent calculus it is relevant on which side of $\vdash$ a formula occurs. In CORE this information is already encoded in the polarities of each formula. We can therefore freely exchange the order of windows in a task. This also motivates the decision to define a task as a *set* of windows. Also note that a task $\Sigma, a^p \rhd a^q$ does not necessarily correspond to an initial sequent in the sequent calculus because the a's might have the same polarity (i.e. $p = q$).

The constraint that no support window of a task must be conditional is important because the content of support windows will be presented to the user as some directly available knowledge that can be used to derive the formula in

the goal window of the task. If the content of the support windows would be β-related to subtrees that lie outside the respective window, then these trees would automatically become conditions for any replacement rule that is generated from this window; that is, the β-related subtrees would represent implicit "knowledge" which will be introduced in form of new proof obligations. We assume that this is less suited for interactive proof construction as the user might want to be able to see whether certain formulas in the context are dependent on further formulas, before applying them in the form of a replacement rule.

As an example consider the formula $(D^+ \Rightarrow^\beta (s = t)^-)^- \Rightarrow^\alpha B[s]^+$. Without the requirement that support windows must be unconditional we could generate the following task for the above formula: $(s = t)^- \rhd B[s]^+$. However, although this task gives the impression that the equation $s = t$ could be used directly to transform $B[s]$ to $B[t]$, this is not the case, as $s = t$ is dependent on D^+ and hence the replacement rule $s \rightarrow < t, D^+ >$, instead of $s \rightarrow t$ has to be used to carry out the transformation.

Because tasks are basically representations of subgoals we next define when a task is closed.

Definition 7 *(Closed task) A task $\Sigma \rhd G$ is* closed *iff there exists a $w \in \Sigma \cup \{G\}$ such that w denotes a proved subtree; that is, w is either $true^+$ or $false^-$.*

The initial problem to derive a goal G from axioms $Ax_1, \ldots, Ax_n$ is represented by the system as a FVIFT for the signed formula $(Ax_1 \wedge \ldots \wedge Ax_n \Rightarrow^\alpha G)^+$. The initial task then contains a window for the goal formula G as the goal formula and one window for each of the axioms as supports. This motivates the following definition of an *initial task*.

Definition 8 *(Initial Task) Let G be a formula and $Ax_1, \ldots, Ax_n$ formulas that represent axioms from which G should be derived. Let further R be a* FVIFT *for $(Ax_1 \wedge \ldots \wedge Ax_n \Rightarrow^\alpha G)^+$, w_i a window on Ax_i and g a window on G, then $w_1, \ldots, w_n \rhd g$ is the* initial task *for G.*

From now on we will not distinguish anymore between a window and the formula it contains when we represent tasks. Hence, the initial window for G with axioms Ax_i will be represented as $Ax_1, \ldots, Ax_n \rhd G$. Note that this implies that we can encounter tasks of the form $\Sigma, A^p, A^p \rhd G$. In this case the A^p are syntactically equal formulas that occur in different windows. However, because we are dealing with windows, rather than formulas we have to treat the A^ps as different entities.

A goal window of type β can be split into two tasks by decomposition of the goal formula. We always keep track of all tasks that are created during a proof attempt with the help of an *agenda*.

Definition 9 *(Agenda) An* agenda *is a set of tasks. An* initial agenda *is an agenda that contains only the initial task for a goal G.*

Tasks in the agenda can be manipulated by decomposition of the goal window, application of a replacement rule to the goal formula, closing of the window on

$$\frac{\Sigma \rhd \alpha(A^{p_A}, B^{p_B})}{\Sigma, B^{p_B} \rhd A^{p_A}}\ \alpha_L \qquad \frac{\Sigma \rhd \alpha(A^{p_A}, B^{p_B})}{\Sigma, A^{p_A} \rhd B^{p_B}}\ \alpha_R \qquad \frac{\Sigma \rhd \alpha((\neg(A^{-p}))^p)}{\Sigma \rhd A^{-p}}\ \alpha_\neg$$

$$\frac{\Sigma \rhd \beta(A^{p_A}, B^{p_B})}{\Sigma \rhd A^{p_A} \qquad \Sigma \rhd B^{p_B}}\ \beta \qquad \frac{\Sigma \rhd \overset{\delta}{\gamma}\ ((\Pi x.F[x])^p) \quad \Pi \in \{\forall, \exists\}}{\Sigma \rhd F[x]^p}\ \gamma\delta$$

$$\frac{\Sigma \rhd G[A^{p_A}]}{\Sigma_1 \rhd A^{p_A} \quad \Sigma_2 \rhd G_1, \ldots, \Sigma_n \rhd G_n}\ Focus$$

$$\frac{\Sigma \rhd G \quad G' = Parent(G)}{\Sigma \rhd G' \qquad Subwindows(G') = \emptyset}\ FocusClose$$

$$\frac{\Sigma, F \rhd G}{\Sigma, G \rhd F}\ Shift \qquad \frac{\Sigma \rhd \diamond}{\emptyset}\ Close$$

$$\frac{\Sigma \rhd i}{\Sigma, i \rhd v_1 \ldots \Sigma, i \rhd v_n}\ Apply(i \rightarrow < v_1, \ldots, v_n >)$$

$$\frac{\Sigma \rhd G}{\Sigma, A^- \rhd G \qquad \Sigma \rhd A^+}\ Cut(A)$$

Fig. 7. The task manipulation rules. The rules α_L, α_R, $\alpha_\neg$, and β are subsumed by the rule *Focus*.

the goal formula, or selection of a different goal window (*Shift*). In Figure 7 we introduce a set of rules that allow us to perform exactly these manipulations. The rules are of type $rule : TASKS \rightarrow 2^{TASKS}$, that is, by application of a rule to a task this task is replaced in the agenda by zero or more tasks. Hence, these rules can only be applied in forward direction. We now investigate each of these rules in turn.

α-decomposition Decomposition of a goal formula of type α is the simplest of the task manipulation rules. It realizes the decomposition of a task with a disjunctive goal formula. In order to decompose a formula $\alpha(A^{p_A}, B^{p_B})$ new windows on the A^{p_A} and B^{p_B} have to be opened and one new window has to be chosen as the new goal window. To be able to select either subwindow as new goal window we define three rules: α_L, α_R, and $\alpha_\neg$.

β-decomposition Decomposition of a task with a goal formula $\beta(A^{p_A}, B^{p_B})$ will lead to a split of the task into two tasks with goal formulas A^{p_A} and B^{p_B} respectively. This is realized through rule β.

β-decomposition requires a little more effort than α-decomposition. The reason is that there must be no conditional support windows in a task. That is, after decomposition of a goal formula $\beta(G_1, G_2)$ the constituents G_1 and G_2 can only become support windows if we make them unconditional when applying

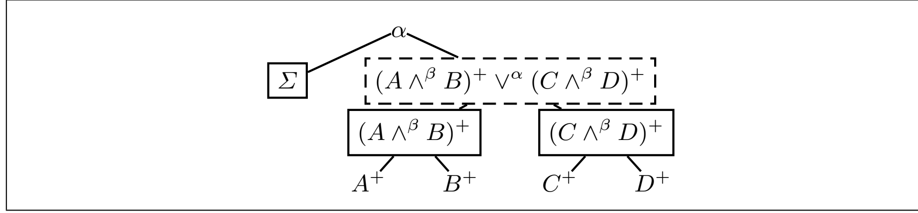


Fig. 8. FVIFT for the task $\Sigma, (A \wedge B)^+ \rhd (C \wedge D)^+$. Dashed lines indicate existing but inactive windows.

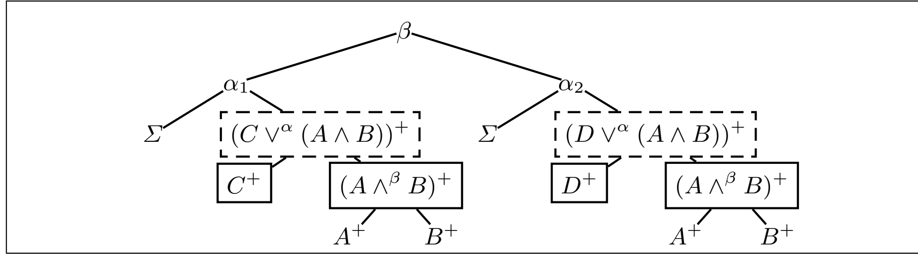


Fig. 9. A logically equivalent FVIFT to the one in Figure 8. This tree would result from the decomposition of the tree in Figure 8 if we realize the β-decomposition with the help of the *Schütte-rule*. The left subtree below α_1 corresponds to task $\Sigma, (A \wedge B)^+ \rhd C^+$ while the subtree below α_2 represents task $\Sigma, (A \wedge B)^+ \rhd D^+$.

β-decomposition. This can be done by splitting up the goal formula $\beta(G_1, G_2)$ while retaining the context φ around it. We can achieve this by applying a rule of the form $\varphi(\beta(A, B)) \to \beta(\varphi(A), \varphi(B))$ which is described in [Sch77] and [Aut03]. Autexier [Aut03] shows that this *Schütte-rule* is admissible in the CORE calculus.

To see why we need the *Schütte-rule*, consider a task $\Sigma, (A \wedge B)^+ \rhd (C \wedge D)^+$. A FVIFT for this task is shown in Figure 8. If we implement the β-decomposition-rule with the help of the *Schütte-rule* we not only change the window structure of the FVIFT as we do with the α-decomposition-rule, but we also change the FVIFT itself. For instance, if we β-decompose the goal window $(C \wedge D)^+$ we obtain the new FVIFT in Figure 9.

We see that the task $\Sigma, (A \wedge B)^+ \rhd C^+$ corresponds to the minimal FVIFT R_1 with root-node α_1 and task $\Sigma, (A \wedge B)^+ \rhd D^+$ is represented by the minimal FVIFT R_2 below α_2 (cf. Figure 9). It is important to note that C^+ and D^+ are unconditional in the respective trees R_1 and R_2. Hence, we can easily make them support windows of a task. This will be important below when we define the *Shift*-rule.

$\gamma-$ and δ-decomposition Focusing on the major subformula of a γ- or δ-formula does not change the set of assertions in the context of this formula. It only has an effect on the variable conditions maintained by CORE.

Compound Decomposition Steps The single decomposition rules above can be combined in a macro-rule that allows us to focus directly on a particular sub-

formula A^p inside the goal window of a task. The uniform types of the nodes in a FVIFT that occur on a path between the selected subformula A^p and the root $G[A^p]$ of the goal window uniquely define a sequence of decomposition steps that need to be applied in order to obtain the chosen formula as a goal window in a single step. The macro-rule *Focus* therefore provides great freedom in the selection of subwindows and simultaneously keeps track about the generated subgoals $\Sigma_2 \rhd G_1, \ldots, \Sigma_n \rhd G_n$ that appear as new tasks in the agenda. Accordingly, the *Focus*-rule can replace the rules $\alpha_R, \alpha_L, \alpha_\neg$, β, and $\gamma\delta$ since they are now subsumed.

FocusClose It is often necessary to undo a decomposition of the current goal window of a task, which has to be realized by closing the focus of the goal window G and with it, all other foci below the parent of G. The *FocusClose*-rule provides exactly this functionality. Note that the *FocusClose*-rule allows us to undo decomposition steps. Closing windows below a node of type α is easy. To close children below a β-node we need to use a reversed *Schütte-rule*.

The Shift-*rule* The *Shift*-rule changes the goal formula of a task. This is particularly important because the rules defined here only allow for manipulation of goal windows. Consider for instance a situation where we have a window on a formula A^p and a window for $(A \Leftrightarrow B)^-$ amongst the support windows for a goal G; that is, $\Sigma, (A \Leftrightarrow B)^-, A^p \rhd G$. If we now want to apply $A \Leftrightarrow B$ in a forward step to A^p we first have to make A^p the goal window. This can be done with the *Shift*-rule.

Because we realized β-decomposition with the *Schütte-rule* we can ensure that goal windows are always unconditional which is a prerequisite for the above definition of the *Shift*-rule.

Closing tasks Tasks are removed from the agenda in case they are closed. This is done by the *Close*-rule which deletes any task $\Sigma \rhd \diamond$ from the agenda, where $\diamond \in \{true^+, false^-\}$.

Replacement Rule Application The way tasks are defined, all replacement rules for a goal window can be generated from the support windows of a task. In the rule *Apply* which applies a replacement rule to a task we to ensure that no "information" gets lost. To see what is meant consider a task of the form

$$\Sigma, G^+, (A^+ \Rightarrow^\beta B^-)^- \rhd A^-$$

The window on $(A^+ \Rightarrow^\beta B^-)^-$ justifies the replacement rule $A^- \rightarrow B^-$. Intuitively, application of the replacement rule to A^- should yield an additional window on B^- such that the task that results from application of this rule is $\Sigma, A^-, G^+, (A^+ \Rightarrow^\beta B^-)^- \rhd B^-$; i.e application of the rule should not remove A^- from the task. However, merely applying this rule to A^- would replace A^- by B^-, which would yield $\Sigma, G^+, (A^+ \Rightarrow^\beta B^-)^- \rhd B^-$. This is not quite what we want since we might need to make use of A^- again.

The solution to this problem lies in the application of the contraction rule to the goal window; that is, we copy the goal window before the application of the replacement rule.

The way the rule is defined here can only be applied to the topmost occurrence i in a goal window and not to subformulas of i. However, this is no problem since we can always focus on a subterm with rule $Focus$ first. Alternatively the rule can easily be extended to operate on subformulas of i as well.

Cut-rule The introduction and speculation of new lemmata plays an important role for interactive theorem proving. For this we introduce the rule *Cut*. This rule allows to introduce a lemma A into the context of a goal G. As a consequence, a new task with the goal to prove that lemma is generated. Implementation of this rule in CORE is quite straightforward as CORE already provides a *cut*-rule which we can use directly at the task level when we update the window and task structure.

3.2 Tasks in Interactive Proof Search

Tasks enable a compact presentation of a proof situation. Instead of listing all replacement rules that can be generated from the context, the support windows give us access to the assertions which we can list line by line on top of the GUI screen, followed by the goal window at the bottom. Such a presentation of a task is meant to resemble the presentation of proofs in mathematical textbooks, where available assertions (axioms and definitions, etc) are mentioned before the theorem is stated[3].

In a system state where the user is shown the current task he has the following options to continue the proof. He can either focus on parts of the goal formula (this includes simple as well as compound decompositions) or apply any other of the task manipulation rules. Application of one of the assertions that are represented by the support windows are realized with the *Apply*-rule by simply clicking on the respective formula. The system then presents all replacement rules that can be generated from the chosen support window. It is then the choice of the user to identify the replacement rule that encodes the preferred application direction of the assertion. This is already an improvement to the situation where the user was presented with all replacement rules for the current window; that is, the replacement rules generated from *all* formulas in the context.

However, it is possible to do even better. For many of the rules that can be generated from a single support window in a task, one can see in advance that they are not applicable in the goal window. For other rules it is likely that they do not represent a suitable application of the formula from which they were generated. As a consequence, the system currently uses a heuristic to generate only those rules that are likely to represent the intended application direction of the assertion (see Section 4).

[3] Of course, in mathematical texts this is often done implicitly or indirectly in the sense that the available mathematical knowledge is referenced by stating the theory or domain context of a proof problem.

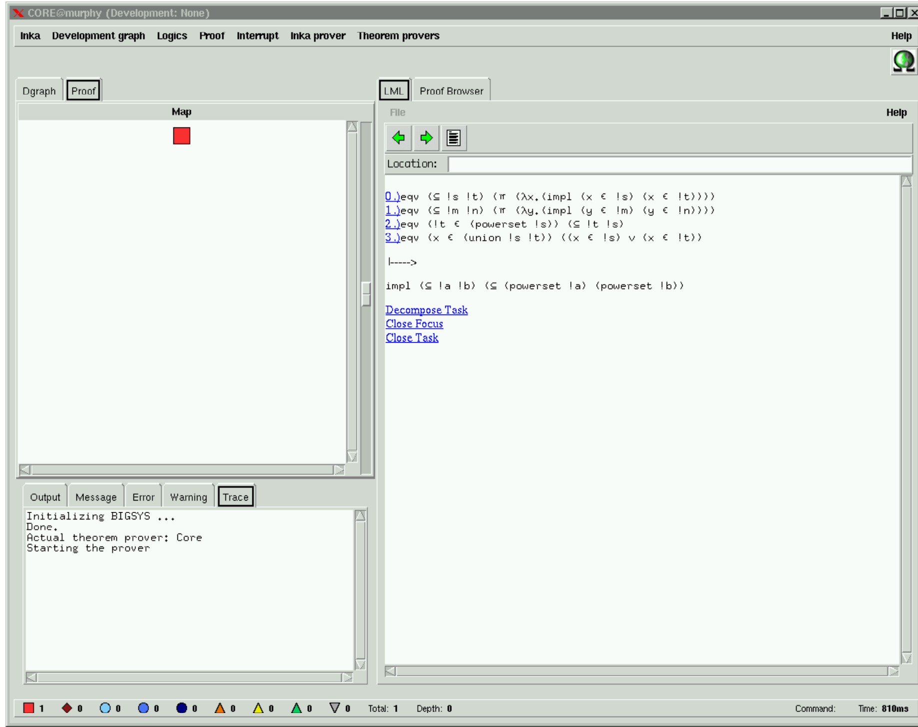


Fig. 10. GUI of the CORE system. The initial task for the example from Section 4 is shown in the window on the right-hand side.

Hübner[Hüb03] describes how replacement rule selection can be supported by the agent-based suggestion mechanism ΩANTS which was until now only used in conjunction with the ΩMEGA [SBF+03] theorem proving system and has now been adapted to this new environment which will provide the basis for the next generation of ΩMEGA.

4 Tasks - A Worked Example

In this section we illustrate interaction at the task layer at hand of an example. The proof problem is given as[4]

Theorem 1 *For sets A and B with $A \subseteq B$ holds that $2^A \subseteq 2^B$. This statement is encoded as:*

$$\forall A, B. A \subseteq B \Rightarrow 2^A \subseteq 2^B \qquad (G)$$

We assume that the following axioms defining $\subseteq$ and 2^M are given:

$$\forall M, N. M \subseteq N \Leftrightarrow (\forall z. z \in M \Rightarrow z \in N) \qquad (Ax1)$$

$$\forall X, M. X \in 2^M \Leftrightarrow X \subseteq M \qquad (Ax2)$$

[4] We use 2^X to denote the powerset of a set X.

4.1 A Natural Deduction Proof

We first sketch a natural deduction proof of the above proof problem which will then be compared to the proof we construct at the task layer. Our natural deduction proof can be divided into 6 different parts as indicated by the boxes B1–B6 below. The conceptual steps addressed in these boxes are indicated by their headlines. Some subproofs employ derived rules (tactics), such as Transitivity of $\subseteq$ or subsequent applications of $\forall_I$ with $\forall_I^*$. Hence, they expand to even larger proofs in a pure natural deduction calculus.

B1 We start with goal G and focus on the right hand side of the implication.

$$\cfrac{\cfrac{\begin{array}{c}[a \subseteq b]^1 \\ \vdots \\ 2^a \subseteq 2^b\end{array}}{a \subseteq b \Rightarrow 2^a \subseteq 2^b}\,\Rightarrow_I^1}{G}\,\forall_I^*$$

B2 Then we apply the definition of $\subseteq$ as given in $Ax1$.

$$\cfrac{\cfrac{\cfrac{Ax1}{2^a \subseteq 2^b \Leftrightarrow (\forall z. z \in 2^a \Rightarrow z \in 2^b)}\,\forall_E^*}{2^a \subseteq 2^b \Leftarrow (\forall z. z \in 2^a \Rightarrow z \in 2^b)}\,\Leftrightarrow_{E_l} \qquad \begin{array}{c}[a \subseteq b]^1 \\ \vdots \\ \forall z. z \in 2^a \Rightarrow z \in 2^b\end{array}}{2^a \subseteq 2^b}\,\Rightarrow_E$$

B3 We focus on the right hand side of the implication of the current subgoal.

$$\cfrac{\cfrac{\begin{array}{c}[a \subseteq b]^1 \\ [c \in 2^a]^2 \\ \vdots \\ c \in 2^b\end{array}}{c \in 2^a \Rightarrow c \in 2^b}\,\Rightarrow_I^2}{\forall z. z \in 2^a \Rightarrow z \in 2^b}\,\forall_I$$

B4 We apply the definition of 2^M as given in $Ax2$.

$$\cfrac{\cfrac{\cfrac{Ax2}{c \in 2^b \Leftrightarrow c \subseteq b}\,\forall_E^*}{c \in 2^b \Leftarrow c \subseteq b}\,\Leftrightarrow_{E_l} \qquad \begin{array}{c}[a \subseteq b]^1 \\ [c \in 2^a]^2 \\ \vdots \\ c \subseteq b\end{array}}{c \in 2^b}\,\Rightarrow_E$$

B5 We apply transitivity of $\subseteq$.

$$\cfrac{\begin{array}{c}[a \subseteq b]^1 \\ [c \in 2^a]^2 \\ \vdots \\ c \subseteq a\end{array} \qquad [a \subseteq b]^1}{c \subseteq b}\,\text{Transitivity-of-} \subseteq$$

B6 We finish the proof by applying the definition of 2^M once more.

$$\dfrac{\dfrac{\dfrac{Ax2}{c \in 2^a \Leftrightarrow c \subseteq a}\forall^*_E}{c \in 2^a \Rightarrow c \subseteq a}\Leftrightarrow_{E_r} \quad [c \in 2^a]^2}{c \subseteq a}\Rightarrow_E$$

4.2 A Proof at Task Layer

At task layer the natural deduction derivations described in B1-B6 collapse into single interactions. We present tasks as in Figure 1, that is, each support window is displayed in one line and the content of the goal window is shown below the supports.

In Figure 10 we can see that the presentation of tasks we use here is very similar to the way they are actually presented in the current GUI of our system. In the GUI we are experimenting with different colored formulas to indicate the polarities of formulas in order to improve readability.

The initial task of our proof problem is:

$$\begin{bmatrix} (Ax1) & (M \subseteq N \Leftrightarrow (z \in M \Rightarrow z \in N))^- \\ (Ax2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \end{bmatrix} \qquad (4)$$

$$(A \subseteq B \Rightarrow 2^A \subseteq 2^B)^+$$

The first step is straightforward: we focus on the right hand side of the implication and obtain

$$\begin{bmatrix} (Ax1) & (M \subseteq N \Leftrightarrow (z \in M \Rightarrow z \in N))^- \\ (Ax2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \end{bmatrix} \qquad (5)$$

$$(2^A \subseteq 2^B)^+$$

where $(A \subseteq B)^-$ occurs now as a new support window. Next we apply the definition of $\subseteq$ to the goal formula $(2^A \subseteq 2^B)^+$. In the GUI this is done by clicking on the assertion formula $Ax1$. In general, the problem of determining the applicable rules involves higher-order unification and is hence not decidable. We are therefore employing a heuristic approach that computes for the selected formula a set of replacement rules that are "most likely" appropriate, that is, we currently employ an imperfect filter. From the suggested rules the user then has to select one rule for application. For the given situation the heuristic computes the single applicable rule (i.e. $(M \subseteq N)^+ \rightarrow< (z \in M \Rightarrow z \in N)^+ >$). The application of this rule results in the new task:

$$\begin{bmatrix} (Ax1) & (M' \subseteq N' \Leftrightarrow (y \in M' \Rightarrow y \in N'))^- \\ (Ax2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \end{bmatrix} \qquad (6)$$

$$(z \in 2^A \Rightarrow z \in 2^B)^+$$

Note that although the application of the replacement rule generated from $Ax1$ has instantiated the variables M and N, we now have an uninstantiated version of $Ax1$ with fresh variable copies M' and N' in the supports. From now on we always assume that after application of an assertion we have an uninstantiated version in the supports, without indicating this explicitely. In the system these fresh copies of applied and thus instantiated assertions are automatically generated by CORE.

In a next step, we apply $Ax2$ to the subformulas $z \in 2^A$ and $z \in 2^B$ respectively. After clicking on the assertion $Ax2$ the system suggests to apply the rule $X \in 2^M \rightarrow < X \subseteq M >$ where we merely have to determine the application position, that is, $(z \in 2^A)$ or $(z \in 2^B)$. We apply it first to the former and then repeat the step and apply to the latter subformula. We obtain the new task:

$$\begin{bmatrix} (Ax1) & (M' \subseteq N' \Leftrightarrow (y \in M' \Rightarrow y \in N'))^- \\ (Ax2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \end{bmatrix} \tag{7}$$

$$(z \subseteq A \Rightarrow z \subseteq B)^+$$

We now focus on the right hand side of the implication in the new subgoal and obtain:

$$\begin{bmatrix} (Ax1) & (M' \subseteq N' \Leftrightarrow (y \in M' \Rightarrow y \in N'))^- \\ (Ax2) & (X \in 2^M \Leftrightarrow X \subseteq M)^- \\ (3) & (A \subseteq B)^- \\ (4) & (z \subseteq A)^- \end{bmatrix} \tag{8}$$

$$(z \subseteq B)^+$$

The rest of the proof consists now in showing (or, if available, applying a lemma for) the transitivity of $\subseteq$.

4.3 Discussion

The previous example illustrates that whole proof blocks in natural calculus are replaced by single and far more intuitive steps at task layer (which are probably in a different order). In summary, we can identify the following three main advantages of the task layer for interactive proof construction:

- It allows to conduct proofs in a way that resembles natural deduction style proofs while at the same time it hides many distracting details of the natural deduction level. Assertion formulas can now be applied to the goal window in a uniform way with the help of simple mouse clicks in the GUI.
- Proofs at the task layer are much shorter and fewer interactions are required. For instance, we do not need to apply quantifier elimination rules because this is done implicitly by CORE. Focusing onto subformulas is realized as single steps, which are ideally realized again in the GUI by simple mouse clicks.

– Tasks allow us to present assertions in an intuitive way to the user. This is a prerequisite for fruitfully supporting the above aspects. In combination with the fact that many branchings of the proof are avoided (cf. backward application the $\Rightarrow_E$-rule) this supports a convenient way to construct proofs.

Further aspects are:

– Since the task level is simply an additional, independent layer on top of CORE soundness of our approach is guaranteed by the underlying CORE system (see [Aut03]).
– Proof methods and automated theorem provers can be easily defined and added to the system to support automated proof construction (see [Hüb03]). This means that tasks can not only be manipulated by application of decomposition and basic rewriting steps but also through application of macro-steps that are described as proof methods.

Work in the direction of intuitive proof presentation and manipulation has been described by Bertot et al. [BKT94]; they show how to focus in a sequent calculus proof to certain subformulas. However, although the approach goes into the right direction it is still very tightly connected to the sequent calculus.

5 Conclusion and Future Work

We have introduced task structures as an intuitive front end for interactive theorem proving based on the CORE system. Our approach is not committed to a particular underlying calculus and it supports very flexible proof construction. This is different to the proof by pointing approach which is strongly connected to the sequent calculus. In Section 4 we have illustrated how explicit decomposition of assertions as required in sequent and natural deduction style calculi is replaced by a uniform application mechanism that avoids decompositions.

In a next step we will now evaluate how well the task transformation rules are suited to model human proofs in the domain of naive set theory. Benzmüller et al. [BFG+03a,BFG+03b] have collected in an experiment on tutorial dialogues with a mathematical assistant system a first corpus of proofs in this domain against which we want to evaluate the naturalness of the task manipulation rules. We also want to investigate whether assertion application at the task layer can be further improved by stronger and better filter; see for instance [VBA03] for ongoing work.

References

[Aut01] Serge Autexier. A proof-planning framework with explicit abstractions based on indexed formulas. In Maria Paola Bonacina and Bernhard Gramlich, editors, *Electronic Notes in Theoretical Computer Science*, volume 58. Elsevier Science Publishers, 2001.

[Aut03] Serge Autexier. *Hierarchical Contextual Reasoning*. PhD thesis, University of the Saarland, 2003. to appear.

[BFG+03a] C. Benzmüller, A. Fiedler, M. Gabsdil, H. Horacek, I. Kruijff-Korbayova, M. Pinkal, J. Siekmann, D. Tsovaltzi, B. Quoc Vo, and M. Wolska. Discourse phenomena in tutorial dialogs on mathematical proofs. In *In Proceedings of AI in Education (AIED 2003) Workshop on Tutorial Dialogue Systems: With a View Towards the Classroom*, Sydney, Australia, 2003.

[BFG+03b] C. Benzmüller, A. Fiedler, M. Gabsdil, H. Horacek, I. Kruijff-Korbayova, M. Pinkal, J. Siekmann, D. Tsovaltzi, B. Quoc Vo, and M. Wolska. Tutorial dialogs on mathematical proofs. In *In Proceedings of IJCAI-03 Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems*, Acapulco, Mexico, 2003.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Thery. Proof by pointing. In *Theoretical Aspects of Computer Science (TACS)*, 1994.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39:572–595, 1935.

[Hua94] Xiaorong Huang. Reconstructing proofs at the assertion level. In Alan Bundy, editor, *Proc. 12th Conference on Automated Deduction*, pages 738–752. Springer-Verlag, 1994.

[Hüb03] Malte Hübner. Interactive theorem proving with indexed formulas. Master's thesis, Fachbereich Informatik, University of the Saarland, 2003.

[Mei03] Andreas Meier. *Proof-Planning with multiple strategies*. PhD thesis, University of the Saarland, 2003. forthcoming.

[PB02] Florina Piroi and Bruno Buchberger. Focus windows: A new technique for proof presentation. In Jaques Calmet, Belaid Benhamou, Olga Caprotti, Laurent Henocque, and Volker Sorge, editors, *Artificial Intelligence, Automated Reasoning and Symbolic Computation*, number 2385 in LNAI, pages 337–341. Springer, 2002.

[RS93] Peter J. Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic Computation*, 3(1):47–61, 1993.

[SBF+03] J. Siekmann, C. Benzmüller, A. Fiedler, A Meier, I. Normann, and M. Pollet. Proof development in OMEGA: The irrationality of square root of 2. In F. Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, Kluwer Applied Logic series. Kluwer Academic Publishers, July 2003.

[Sch77] K. Schütte. *Proof Theory*. Springer Verlag, 1977.

[Smu68] Raymond R. Smullyan. *First Order Logic*. Springer, 1968.

[VBA03] Quoc Bao Vo, Christoph Benzmüller, and Serge Autexier. Assertion application in theorem proving and proof planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, 2003. (poster description).

[Wal90] Lincoln A. Wallen. *Automated Proof Search in Non-Classical Logics. Efficient Matrix Proof Methods for Modal and Intuitionistic Logics*. MIT Press, Cambridge, Massachusetts;London, England, 1990.

Improving the PVS User Interface

Joseph R. Kiniry[1] and Sam Owre[2]

[1] Computing Science Department
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands
kiniry@cs.kun.nl
[2] SRI International
Computer Science Laboratory
333 Ravenswood Ave.
Menlo Park, CA 94025
owre@csl.sri.com

Abstract. We describe a set of new features, whose design is partially motivated by the features of modern programming environments, that has been developed for the PVS theorem prover. We also discuss in detail how these features can be added to other theorem proving environments that use Emacs as a front-end. The major contribution is an integrated lexer and parser (constructed with the *semantic* Emacs package) for the PVS language that makes Emacs context-aware. This framework automatically enables a broad set of new functionality including the ability to browse PVS constructs (declarations, theories, types, proofs, etc.), quick access to construct definitions in three new ways (a short-cut sidebar, menus, or implicit hyperlinks), contextual help, and context- and type-aware completion. Other new features discussed include the ability to visually expand and collapse structured elements of specifications and sequents, the graphical representation of language elements (with little impact on the ability to cut-and-paste terms), and a user-extensible, type-aware pretty-printer that has been added to the PVS prover.

1 Introduction

The user interfaces (UIs) of modern, popular theorem provers provide all the core functionality necessary to write specifications and perform proofs. Unfortunately, most of these features seem to focus on the *mathematics*, rather than the *mathematician*.

Only the most advanced UIs provide features approximating those of the rudimentary programming environments of a decade ago (e.g., symbol completion, syntax highlighting, and basic documentation lookup). As a result, theorem prover users, especially those that are also prolific programmers, have begun to demand many of the features of modern integrated development environments (IDEs).

Because the design of most theorem proving environments decouples the UI from the prover, there is some "modality impedance" between the two domains. For example, the UI is unaware of the user's intent in all but the most general fashion. This impedance exists because nearly all of the "intelligence" of these tools is in the prover, rather than the UI.

We suggest that the generic facilities, both conceptual and technical, of advanced programming environments (like those available for working with programming languages like C++ and Java) are applicable to the UIs of modern theorem provers. To that end, this paper discusses our initial work in improving the user interface of PVS. Our subgoal is not to force the user to work in a new way, but only provide a generic set of new features that can help users write specifications and perform proofs.

Our primary contribution is that we have made the PVS UI aware of the *semantic context* in which the user is working. The PVS UI has been augmented by adding a front-end lexer and parser for the full PVS language. This infrastructure represents high-level PVS constructs (declarations, theories, types, proofs, etc.) in a generic fashion. This generic representation enables the use of a wide range of new tools previously available only to traditional programmers using modern Integrated Development Environments.

The new functionality available through this parser-based approach includes a construct browser, quick access to construct definitions via a short-cut sidebar, menus, or implicit hyperlinks, contextual help, and context- and type-aware completion.

We have also added other new functionality to both the PVS UI as well as the prover itself. In addition to the previously mentioned ones, the UI has two new features. First, we have added the ability to visually expand and collapse structured elements of specifications and sequents. Second, language elements can be represented graphically rather than textually, but with little impact on the ability to cut-and-paste terms. Finally, a user-extensible, type-aware pretty-printer has been added to the PVS prover.

1.1 Related Work

Other environments provide some of the features that we have added to PVS. To our knowledge, none provide all of them as a coherent unit, nor do any use an integrated parser to enable the features we discuss herein.

Mathematics Environments Mathematics environments provide interactive UIs. Most theorem proving environments come from a bare-bones tradition. Most effort is spent focusing on the mathematical rather than visual infrastructure. Until recently, with the growing adoption of Proof General [2], interfaces were nothing more than glorified command-lines [15].

We regard Proof General and PVS as the most widely used environments with the most advanced UIs. Both use Emacs as a front-end, so much of their power is simply a side-effect of that choice. Features like syntax highlighting,

completion, and hypertext documentation are examples of such pleasant side-effects. Neither Proof General nor, obviously until now, PVS has the features that we discuss in the following pages.

Some theorem proving environments like Jape and CtCoq have been used as UI/HCI testbeds [4, 5, 15]. Many of the innovations that were originally introduced in specialized environments have now found their way into general purpose front-ends (e.g., proof-by-pointing in Centaur and CtCoq is now available in ProofGeneral, the PPML-based layout and pretty-printing facilities of CtCoq are partially functionally reproduced in some of this work, etc.).

General purpose computation environments, particularly quality commercial environments like Mathematica, Maple, and Matlab, have rich UIs. It is not uncommon to provide WYSIWYG-ish interfaces that use mathematics fonts and provide direct editing of terms with a point-and-click, drag-and-drop UI. While we are not attempting to duplicate such fancy UIs, we should note their visual features are compelling and inspirational, but their editing features and flexibility are actually inferior to most sophisticated environments like those we mentioned above. We plumb the features of these commercial environments like any other quality pool of UI ideas.

Coupling an advanced UI like that of Mathematica to a prover like PVS is an interesting experiment [1]. But, as neither tool was designed as a reusable component, such coupling would be difficult. Instead, we try to learn from these advanced mathematics environments, borrowing the features that we find most useful in our day-to-day work.

Programming Environments Modern programming environments[3], contrary to theorem proving environments, provide a large, rich set of UI features. Common features that are not specific to traditional (non-mathematical) programming (that is, they might be applicable to mathematical systems like theorem proving environments) include code browsing, automatic code and documentation formatting, type-aware and template-based completion, general project management, and integrated help, API documentation, process tracking, and revision control support.

Many currently popular environments have recently added refactoring functionality [7, 14]. Refactoring is the process of changing the implementation of a system in order to improve some aspect of the implementation while preserving its capabilities. Examples of refactoring include safe renaming of program features (variables, functions, methods, etc.), function extraction and insertion, and safe modification of feature visibility (e.g., making a public variable private, automatically writing getter and setter methods, and automatically inserting these methods at all program locations that access the original public variable).

[3] Some of the tools with which we have experience and from which we borrow ideas include Borland's jBuilder and Together ControlCenter, Eclipse, Eiffel Studio, IntelliJ IDEA, jEdit, Metrowerks CodeWarrior, NetBeans, Oracle9i JDeveloper, Sun ONE Studio, and the various "Visual*" tools (i.e., Microsoft's Visual Studio, IBM's VisualAge, WebGain's Visual Café).

We believe that each new feature, after it has seen moderate success in the mainstream programming community, should be evaluated as a potential addition to theorem proving UIs. This work is the initial result of such an evaluation.

While we have evaluated features from over a dozen modern IDEs, we should note that the tool with which we have the most experience, and to which we always return after experimenting with these other environments, is, of course, Emacs.

2 The Semantic Package

Adding the new capabilities we had envisioned to the PVS UI requires one either (a) write PVS-specific functionality for Emacs or (b) rely upon a generic language-based framework that enables the use of existing and future tools based upon the framework. With an eye toward saving time, increasing reliability, and taking advantage of the hard work of other Emacs enthusiasts, we choose option (b). And, after evaluating the various generic framework's available for Emacs, we chose to use the semantic package.

The semantic package (a collection of related code) for Emacs is used to build lexers and parsers in Emacs Lisp (elisp) [12]. To use the semantic package with a new language, thereby enabling all semantic package-based tools (which we discuss at length in Section 4), one must write a grammar specification of the language. We discuss the process of adding a new language to the semantic package in Section 3.

The semantic package includes, at its core, a lexer generator, and a compiler compiler, or what is known as a *bovinator* in the semantic package's nomenclature. The core utility is the *semantic bovinator* which has similar capabilities to the GNU `bison` compiler compiler [6]. Since the semantic package was not designed to be as feature rich as these tools, it uses the term "bovine" for cow, a lesser cousin of the yak and bison.

Recent versions of of the semantic package (mid-2003) include a second compiler compiler called *wisent* (the European bison), which is effectively a full GNU `bison` implementation in elisp. The work described in this paper does not yet use wisent.

We chose to use the semantic package because once a language has been added to the it, many powerful tools become available for use. Essentially, by relying upon a generic foundation, we enable

3 Adding Support for the PVS Language to the Semantic Package

To enable all of the semantic package-dependent tools available for Emacs we must write PVS language support for the semantic package. First, a PVS lexer must be created that properly tokenizes the PVS language to a format that the semantic package parser understands. Then, a PVS parser must be created that

generates an abstract representation (an s-expression that represents a parse tree) of the lexed input in a form that the semantic package's tools understand.

This section covers the details of this process, thus only need be read by developers of Emacs-based theorem proving environments who wish to enable semantic package-based tools for their environments. General readers interested in the new features of the PVS UI should feel free to skip to Section 4.

3.1 Lexing

The first step of parsing is to break up the input file into its fundamental components, known as tokens. This process is called *lexing*. The output of a lexer is a sequence of *tokens*.

Token Specification Each language has a core set of structures that must be specified to construct a lexer. First, every language has a set of *keywords*, symbols with special meaning within the language. Second, basic types like identifiers, strings, characters, and floating point and integer numbers all have a standard representation. Finally, comments have a particular structure.

Other syntactic tokens that need be considered include quoted characters, opening and closing parenthesis-like constructs (e.g., ‘(’, ‘)’, ‘{’, ‘}’, ‘[’, ‘]’, etc.), comment prefix and suffixes, newlines, punctuation, strings, symbols, and whitespace. A few of these constructs have particular importance in some languages, so we will examine them in more detail.

Values. The structure of basic values like numbers are specified in the semantic package using regular expressions. For example, numbers in PVS are simply sequences of the digits 0 though 9. The default number lexer in the semantic package will lex values like those in the C and Java programming languages (numbers with an optional minus sign, in decimal, hexadecimal, and octal, floating point numbers, numbers in exponential notation, etc.), which is inappropriate for PVS number value parsing. The structure of PVS's numbers are specified by setting the variable `semantic-number-expression` to value “`(<[0-9]>+)`”. This regular expression means “match any contiguous sequence of the digits 0, 1, . . . , 9 as a single token”.

Symbols. The default definition for a symbol in the semantic package is a sequence of characters and underscores. PVS's symbols can include question marks as well, thus the definition of symbol is also replaced.

Punctuation. The notion of “punctuation” in the semantic package is defined as “any token that is exactly one character long”. PVS has a large set of special symbols that are more than one character long, but because of this limited definition of punctuations, they must be parsed by basic grammar rules rather than handled within the lexer.

For example, one would normally consider the PVS operator "`##`" as punctuation, but instead it must be parsed with the trivial rule "`HASH_HASH : HASH HASH ;`" where `HASH` is the token '`#`'.

Parenthesis-like operators also must be single characters. Thus, PVS's multi-character operator pairs like "`{|`" and "`|}`" must be lexed in the same fashion.

Whitespace. Whitespace is usually one of those things that one can ignore when parsing. Unfortunately, some programming languages, like Python, given semantics to whitespace [20]. In the case of Python, nesting level and scope are determined by whitespace indentation. The semantic package can handle such situations via a built-in flag that changes the manner in which the lexer handles whitespace. Fortunately, whitespace in PVS can be wholly ignored.

Comments. The final issue that must be addressed is comment specification. If comments are not important to parsing (i.e., they have no semantics), then the semantic package's lexer must only know how comments start and end. Otherwise, we must specify to the semantic package that there are no comments in the language being lexed and write production rules for the grammar of comments. We currently use the former strategy when parsing PVS, noting to the semantic package that PVS comments begin with the '`%`' character and continue until the end of the line.

Case Distinction. The default behavior of the semantic package's lexer is to match keywords in a case-sensitive fashion. This behavior is controlled by the variable `semantic-case-fold`.

PVS is a strange language with regards to case-sensitivity (partially by design, and partially by the case-sensitivity features of Allegro 6). In PVS 3, keywords are case *insensitive*, but all other symbols are *case sensitive* [16]. Currently, we lex in a case-sensitive fashion and provide a function that will indicate if any keywords are used in a non-standard lowercase or mixed-case fashion.

Semantic Lists The semantic package must understand parenthesis-like structures because it handles lexing of such structures specially. When lexing takes place, a "depth" is specified. If depth is not specified, then all parenthesized sequences of tokens are elided into *semantic lists*. That is, every sequence of input symbols "(t0 t1 ... tk)" is represented by exactly one token.

This elision is done for two reasons: speed and refinement. Ignoring all tokens between pairs of parenthesis-like constructs means that the lexer can be very fast. Low level structures like mathematical expressions, and medium-level structures like delimited code blocks (e.g., curly braced-delimited code blocks in most Algol-like programming languages) are simply not important to understanding the high-level structure of a document—and understanding such structure is the whole point of the semantic package.

When the depth of parsing is specified (by a non-negative integer value), then all semantic lists up to and including that nest depth are lexed. That is to say,

if depth is 2, then every top level parenthesized expression e, and any outermost parenthesized expression contained in e, is lexed. All deeper expressions continue to be lexed as semantic lists. Such a lexing strategy lets us refine lexing depth on demand, only lexing (and thus parsing) the details of an input file as necessary.

This combination of refinable, fast lexing is necessary because, as we will discuss later, the semantic package's lexer and parser is often running continuously while a user edits a file.

Building a Lexer A "major mode" in Emacs is a set of customizations specific to editing text of a particular sort. For example, major modes exist for editing Java, C, LaTeX, Texinfo, etc. Happily, the PVS system comes with a basic major mode for the PVS language. Thus, most of the work is done for us.

Building a lexer with the semantic package can be quite easy. If Emacs supports the input language with a "major mode", then the job is trivial. If a major mode does not exist for the input language, a basic one can be written in just a few lines of elisp.

The semantic package uses the major mode's syntax table (a definition of which characters are opening deliminters, which are parts of words, which are string quotes, etc.) for the basic definition of language tokens. This information is augmented by the grammar author via an explicit specification of keywords and punctuation.

For example, for the PVS language we have the following specification:

```
%token AND          "AND"
%put   AND           summary
           "Binary logical operator; typically conjuction:
            <expression> AND <expression>"
...
%token LEFT_BRACE   open-paren "{"
%token RIGHT_BRACE  close-paren "}"
%token SEMI_COLON   punctuation ";"
```

The first line specifies that the token "AND" is represented in BNF by the shorthand symbol `AND`. Additionally, attached to this symbol is a string that is tagged as a "summary". (We'll discuss summaries in Section 4.1.) Left and right curly braces are specified as open and close parenthesis-like structures with the `open-paren` and `close-paren` token types. Lastly, semi-colon is specified as punctuation.

The structure of the values of the basic types of the input language must be specified to the semantic package's lexer so that values will be lexed properly. For example, if strings are delimited with unusual characters, or as discussed earlier, numbers are not written as in the C programming language, extra work is necessary to denote such.

Once all of these definitions are written, the lexer is complete. There is no explicit "create lexer" function, it is encapsulated in the parser construction discussed in the sequal.

The primary entry point for the lexer is the `semantic-flex` function. Normally, it is never called by a user, as it is usually called by the parser automatically. Only when debugging the lexer is it necessary to manually call this function. See Section 3.3 for details.

3.2 Parsing

To parse a language with the semantic package a BNF specification, much like those written for use with the GNU `bison` tool, must be written. This specification has all the standard parts of a compiler compiler input file: a prelude, a description of terminals and non-terminals, production rules, etc. Production rules for the semantic package are written in elisp. The BNF specification is compiled using the `bovinate` function into a parse table. This table is used to drive parsing; the BNF is simply a convenient input format for the grammar author.

For some languages it is difficult, if not impossible, to write a BNF grammar. In such situations, contextual information can be collected during parsing to help make parsing decisions, or alternative parser infrastructures can be used to generate parse trees. For example, the semantic package includes a regular-expression-based parser for the Texinfo documentation language, whose grammar cannot be specified with BNF [12].

The semantic package comes with BNF grammars for the C, C++, Emacs Lisp, Erlang, Java, Scheme, and Makefile languages. It comes with wisent grammars for awk, calc (a tutorial), Java, Simula, Python, and the semantic package's input language itself. Basic support for parsing the Texinfo and LaTeX languages is also available.

Language Settings Several settings are available to let the language grammar author specify properties of the language that is being parsed by the semantic package. All settings start with the '`%`' character. We only mention a few of the more important settings here.

The `%start` setting specified the top-level production rule of the grammar. The `%outputfile` setting dictates where the grammar generated from the BNF input file should be written. Setting `%parsetable` is used to indicate the name of the parse table that is generated from the BNF.

Tokens are specified with the `%token` keyword. Each token has a name (an elisp symbol), a type, and a textual representation. The textual representation is what is matched during parsing (in either a case-sensative or case-insensitive fashion, as discussed in Section 3.1) to recognize the token. The token type is one of `open-paren`, `close-paren`, or `punctuation`. Examples of token declarations were shown in the preceeding section. Tokens without a type are keywords.

The `%put` setting attaches an arbitrary set of symbol-value pairs to a token. Specific properties, like the `summary` property shown in the earlier example, are used in special ways by various semantic package tools. See Section 4.1 for a discussion of how `summary` is used in particular.

The semantic package builds a parser that understands basic scoping rules. Most languages have a character or a short string that separates the names of scopes. For example, in Java the period character separates package names. In PVS, the period character is also used as the scope name separator. This information is supplied to the semantic package via the `semantic-type-relation-separator-character` variable. Thus, for PVS this variable is set to '`.`', which informs the semantic package that each period used in a identifier signals the use of a new nested context. This hint is used during completion to determine the appropriate scope for the completion match.

The important constructs of a language are indicated with the setting `semantic-symbol->name-assoc-list-for-type-parts`. For an object-oriented programming language like Java, the set of constucts includes classes, variables, methods, imports, and packages. For PVS, our core set of constructs is modules, theories, variables, constants, parameters, axioms, postulates, assumptions, formulas, and types[4]. We are also considering adding conversions and auto-rewrites to this set. There is a supplimentary set of constructs that is not used for shortcuts or menu generation (both of which are discussed later), but are used for features like completion. The supplimentary constructs are libraries, importings, and exportings.

Parse Rules The semantic package's parse rules are very similar to those of the `yacc` and `bison` input formats. Each rule is of the form

```
RESULT : MATCH1 (optional-lambda-expression1)
       | MATCH2 (optional-lambda-expression2) ;
```

`RESULT` is a non-terminal symbol and each `MATCH` is a list of elements that are to be matched if `RESULT` is to be made. These elements can be tokens, text, regular-expressions, or non-terminals.

The optional lambda expressions are the production rules of the grammar. If `MATCH1` is matched, then the production rule represented by `optional-lambda-expression1` is executed. These elisp expressions are used to build the parse tree, track stages in the parse, or do whatever else the grammar developer desires as they are arbitrary lambda expressions.

Optional Lambda Expressions The optional lambda expressions (OLEs) used in the semantic package's production rules have a short-hand syntax that is designed to simplify the specification of standard parse tree substructures. For example, an OLE of the form "`( $1 )`" results in a lambda return which consists entirely of the string or object found as the first element of a match. Functions can, of course, be called in OLEs. The OLE "`( (foo $1) )`" executes the function `foo` on the first match of the rule and returns the result.

[4] We differentiate postulates from formulas in the same manner as the PVS prelude: postulates are provable by decision procedures. Types include datatypes and co-datatypes.

A comma character is used as shorthand in OLEs to denote catenation. So, the OLE "`( ,(foo $2) )`" executes `foo` on the second match of the rule and splices the result into the return list of the OLE. An apostrophe is used to denote list construction in a similar manner. The OLE "`( '(foo) )`", for example, calls the function `foo` with no arguments and puts its result into a list, and splices that list into the return list of the rule.

Helper Functions. Three helper functions are also available for OLEs.

An OLE of the form "`(EXPAND token nonterminal depth)`" performs a recursive parse on the token passed to it. This token is often a semantic list, as discussed in Section 3.1. The parameter `nonterminal` is the name of the parse rule from which the parser will start this recursive subparse. The `depth` parameter, which must be an integer value of at least 1, specifies how far the parser should descend into semantic lists during the expanded parse.

The helper function `EXPANDFULL` is just like `EXPAND` except that the parser will iterate over `nonterminal` until there are no more matches. This simplifies a common case and helps with error skipping during subparsing.

Finally, the `ASSOC` helper function is used by writing OLEs of the form "`(ASSOC symbol1 value1 symbol2 value2 ...)`". This expression is used to create an association list where each `symbol` is in the list if the associated `value` is non-`nil`. The created structure is a standard list of dotted pairs.

Translation Strategy for PVS Rules There were two options for writing a BNF grammar for the PVS language. First, there is a BNF specification available (via the function `help-pvs-bnf`) of the top-level constructs in the PVS language in the built-in help for PVS. Alternatively, the specification used to generate the actual PVS parser is available in the source distribution of PVS.

Rewriting EBNF. Either of these sources has to be significantly reworked because the semantic package's BNF parser does not handle Extended BNF, thus does not support repetition constructs like '`+`' and '`*`'[5].

Because we wanted to gain significant experience with the semantic package and we wanted to be able to parse the full PVS language (not just the top-most forms), we chose to reformulate the full language specification from the source. To deal with the lack of extended BNF, we reformulated all EBNF terms in the following regular fashion.

– Terms of the form `rule_foo?` (zero or one applications of foo) are rewritten as

```
optional_rule_foo : rule_foo
                  | EMPTY ;
```

[5] This is an unfortunate omission from the Semantic package version 1.4 that is remedied with the new *wisent* parser framework in the upcoming Semantic package version 2.0.

where term EMPTY matches the empty input.

– Terms of the form rule_foo+ (one or more applications of foo) are rewritten as

```
one_or_more_rule_foos : rule_foo one_or_more_rule_foos
                      | rule_foo ;
```

Note that the longer match proceeds the shorter match in this rule. This is necessary because the rule framework matches rules in the order that they are specified.

– Terms of the form rule_foo* (zero or more applications of foo) are rewritten as

```
zero_or_more_rule_foos : one_or_more_rule_foos
                       | EMPTY ;
```

This means that all rules that are used in the starred form necessitate writing a complementary one-or-more rule.

Lists. To parse mark-separated lists (e.g., comma separated lists) we rewrite rules in a regular form as well. A rule written in the pseudo-EBNF of the PVS source grammar of the form "{rule_foo ++ ','}", which means "one or more comma separated matches of the rule rule_foo", is rewritten as

```
one_or_more_comma_separated_rule_foos :
     rule_foo COMMA one_or_more_comma_separated_rule_foos
   | rule_foo ;
```

A similar obvious structure holds for rules of the form "{rule_foo ** '.'}":

```
zero_or_more_period_separated_rule_foos :
    one_or_more_period_separated_rule_foos
  | EMPTY ;
```

where one_or_more_period_separated_rule_foos is like the above example with COMMA replaced by PERIOD.

Parenthesized Expressions. Finally, we write parenthesized expressions in a standard form. A rule of the form "(rule_foo)" is written as

```
parenthesized_rule_foo : LEFT_PARENTHESIS rule_foo RIGHT_PARENTHESIS ;
```

The use of all of these regular forms result in a grammar that is longer than it need be because all constructs are explicit and the grammar is not collapsed to minimal form. The tradeoff for this increase in grammar length is that the semantics of every rule is completely clear by name. As a result, the semantic package's PVS grammar has just over 300 rules while the original Ergo grammar has around 110 rules.

Parse Trees All semantic package-based tools are designed to work with parse trees that have the same basic (but extensible) format. That is, all input languages scanned by the semantic package's parsers generate parse trees with the same basic structure. The benefit of this approach is that tools that are based upon the semantic package are language-neutral; e.g., the semantic package-based code browser works equally well with C++ as with Java.

In general, all tokens returned by the parser are of the form

```
("NAME" TYPE-SYMBOL ... "DOCSTRING" PROPERTIES OVERLAY)
```

`NAME` and `TYPE-SYMBOL` are the only mandatory components. `NAME` is a string that represents the nonterminal, usually a named definition that the language will use elsewhere as a reference to the syntactic element found.

`TYPE-SYMBOL` is a symbol representing the type of the nonterminal. Valid type symbols are any elisp symbol.

`DOCSTRING` is a required slot for a nonterminal, but can be `nil`. This slot is used to summarize the nonterminal in the parse tree. Many languages have documentation written in a comment nearby a declaration. In these cases, `DOCSTRING` is `nil`, and the function `semantic-find-documentation` is used to find the documentation instead.

`PROPERTIES` is a slot generated by the semantic package's parser harness and is not provided by the grammar author. This slot's values includes the properties mentioned earlier that are specified via the setting `%put`, discussed in Section 3.2.

Finally, the `OVERLAY` property specifies the location of the token—its beginning, ending, and buffer.

Some default token types have an extra slot called `EXTRA-SPEC` that is used for common extra specifiers. These specifiers provide optional additional details which are used by various semantic package-based tools. Examples of such constructs are: information about type hierarchy, the number of levels of dereferencing in a variable, the number of levels of pointers in a variable, the type modifiers of a type declaration, const-ness of an expression (i.e., whether an expression is constant or not), exception signatures for languages like Java, constructor and destructor identification, etc.

Standard Tokens The standard tokens for nonterminals for functional languages are of types *variable*, *function*, *type*, *include*, and *package*. Each has a specific structure which semantic package-based tools expect. We reuse the first and third standard token types for the PVS grammar, as PVS has variables and types. We map basic types *function* to *formula*, *include* to *import*, and *package* to *library* to match the PVS vernacular, but each construct's structure is unchanged from that of the default.

For example, a semantic package *type* declaration token has the form

```
("NAME" type "TYPE" (PART-LIST) (PARENTS)
        EXTRA-SPEC "DOCSTRING" PROPERTIES OVERLAY)
```

A `type` token is used, for example, to represent C++ structures, unions, enumerations, typedefs, and classes. They are used to represent PVS types and

(co)datatyes. NAME is the name of the type, and TYPE is the classifier of the type (e.g., "class"). PART-LIST is a list of entries that are found inside of compound types (e.g., the fields of a structure). PARENTS is used for two purposes: it is either a list of parent types (used if the language has a notion of inheritance and TYPE is a type that exhibits inheritance), or, in the case of an alias type, it is the type that is aliased. All the other fields have been discussed previously.

PVS Rules Here is a small excerpt from the PVS semantic package BNF grammar. The rule in the original PVS grammar source file that describes how to parse a PVS theory is as follows:

```
theory ::= 'THEORY' [exporting]^e 'BEGIN' [assuming-part]^a
            [theory-part]^t 'END' id
              <module([noexp()|exporting]^e,
                      [noass()|assuming-part]^a,
                      [notheory()|theory-part]^t,id)>;
```

This rule is rewritten in the following form for the semantic package.

```
theory : THEORY optional_exporting
         BEGIN
         optional_assuming-part
         optional_theory-part
         END symbol
         ( (progn
                (setq semantic-current-pvs-theory (car $7))
                (clrhash semantic-current-pvs-theory-formulas)
             ((car $7) theory $2 $4 $5 nil)) ) ;
```

This match should be nearly self explanatory but for the OLE. The symbols that are capitalized are tokens, e.g., THEORY is the token "THEORY". The first line of the OLE is tracking the current theory that is being parsed by storing its name in the variable semantic-current-pvs-theory. This information is used during parsing in a variety of ways, but primarily as a hash into a table that stores information about the formulas of a theory. That table is cleared in the second line of the OLE. The final line of the OLE returns the theory token that represents the full parse tree of the PVS theory that has been parsed by this rule.

3.3 Debugging

Debugging a lexer and parser for the semantic package can sometimes be difficult. Although debugging facilities are provided, they are very rudimentary.

Debugging the Lexer As mentioned earlier, the primary entry point for the lexer is the semantic-flex function. To debug the lexer, the easiest thing to do is to manually call this function in a buffer that you wish to lex and examine the returned list of tokens. As the lexer is automatically generated from information

in the major mode, and given that the semantic package's lexers are generally quite simple, this manual debugging process is usually sufficient to determine that the lexer is correct.

Debugging the Parser The only interface to debugging a parser within the semantic package is the function `bovinate-debug`.

While debugging, two windows are visible within Emacs. One shows the file being parsed. In that window the syntactic token being tested is highlighted. The second window shows the BNF source and the current rule being matched is indicated with the cursor.

Additionally, in the minibuffer, a brief summary of the current parse context is shown. The initial part of the summary is a dotted pair of the form "`(TYPE START . END)`", indicating the current type being parsed and its beginning and end positions in the source buffer. The rest of the display is a list of all strings collected for the current rule thus far.

When all matches of the current rule fail, normally the parser restarts with the top-level rule. This makes semantic package parsers robust, as they must perform best-effort parsing while a user is editing a file. The debugger, on the other hand, exits in this situation. It can also be exited by calling `keyboard-quit`.

We suggest debugging grammars in a bottom-up fashion, testing various subtrees of the grammar independently. For example, expression parsing was tested prior to formula parsing in PVS.

4 Capabilities Enabled by the Semantic Package

Once we have added our new lexer and parser to the semantic-package for our language as described in Section 3, a number of useful features are automatically available in Emacs.

A comprehensive elisp API is also available for the semantic package to let a tool author query and manipulate a parse tree in a large number of ways. As such programming is not the focus of this paper, we will just discuss the pre-built tools that are available for semantic package-enabled system like our improved PVS UI.

4.1 Built-in Generic Functionality

The two main pieces of "built-in" functionality are contextual help and smart completion.

Contextual Help The summary information `%put` on tokens, as discussed in Section 3.1 is used for contextual help. When the cursor is on a token, or within the block associated with a token for more than a specified period of time (usually a few hundred milliseconds), the specified help message is displayed in the minibuffer.

This functionality is used to document the grammar of constructs (a kind of context-aware `help-pvs-prover-commands`) as well as the common usage of operators like the conjunction example earlier in this paper.

Scope-aware and Type-aware Completion Normal completion in Emacs (activated via the `dabbrev-expand` function and its relatives) dynamically searches backwards from the current point to the preceding word for which the current word is a prefix. This works well in practice, but is completely ignorant of the syntax and semantics of the language in which the user is working.

Modern programming environments offer scope- and type-aware completion. These IDEs parse and typecheck the input file as the user edits. When completion of a construct (e.g., a type, variable, or method name) is requested, a list of all legitimate (type-correct and visible within the current scope) alternatives is shown.

Unfortunately, because of PVS's heavy use of overloading and parameterization, we have not yet reached this level of type-aware completion in our new PVS UI[6]. At this time, when a completion is requested, only those constructs that are visible in the current scope and *potentially* type-correct are shown. Completion choices are shown in a scope-dependent order, with matching declarations in inner scopes shown before those in outer scopes.

4.2 Semantic Package Aware Tools

Three tools are currently available that are semantic package aware as well. All three of these tools work with our new PVS UI. A PVS user need not take advantage of all of these tools (for example, the authors do not care for menus) as they can be enabled and disabled independently.

Additionally, because these are *language-generic* tools, some readers who are Emacs users are probably already familiar with them. Most importantly, gaining experience with any of these tools is leveraged across all languages supported by the semantic package. This represents an excellent investment of time-for-utility on the part of Emacs power-users.

Essentially, each of these three tools offers different ways to organize the same set of information. The fact that three choices are available today is a good thing because different users work in different ways.

We emphasize that all future tools built with the semantic package will automatically work with our new PVS UI.

Imenu The Imenu facility is built into recent versions of Emacs. It offers a way to find the major definitions in a file by name, via a nested set of menus. Any readers who have used the AucTeX or Java Developer's Environment (JDE) packages have possibly used Imenu many times without knowing it.

[6] Currently the semantic package-based parser front-end is not communicating with PVS's typechecker, but we plan on adding this feature in the future.

The PVS user can now specify with Emacs's *customize* feature which PVS constructs (possibly none) are shown in a summary menu. By default, all constructs are shown in a hierarchical menu, organized by theory.

Speedbar Speedbar is a package for Emacs that summarizes information about the current buffer's context. The original inspiration is the "explorer" often used in modern development environment, office packages, and web browsers.

In the speedbar frame or window, a tree view is show of the current buffer. A node in the tree can be expanded or contracted with a click of the mouse or with a keypress. When a construct in the summary tree is activated (again, with a mouse click or keypress) the point (cursor) in the current buffer jumps to the corresponding element in the source file.

Once again, the PVS user can customize which PVS constructs are summarized in their speedbar.

ECB – The Emacs Code Browser The Emacs Code Browser is a source code browser for Emacs. It is a global minor-mode which displays a (user customizable) set of windows that can be used to browse directories, files, and file contents (i.e., the substructures identified during parsing with the semantic package).

The ECB has several very nice features. First, if you an arrange the frames and windows of your working environment in a manner that works best for you, ECB will remember their positions automatically. Second, one of the windows available in ECB is a "history" of the buffers that you most recently visited. Third, you can indicate to ECB the paths relevant to your current project and it will automatically include those in its speedbar-like built-in frame. This means you do not have to wade through the filesystem to find that one file that defines a theory that you need. Instead, it is available with just one or two clicks of the mouse.

We have discussed only some of the features of Imenu, Speedbar, and ECB. The reader is encouraged to discover about these excellent Emacs packages even if they are not PVS users. See [3, 13] for more information about ECB and Speedbar. Imenu is a built-in part of all modern Emacsen, so see your Emacs info documentation for more details on it.

4.3 New Functionality

Two pieces of functionality that combine the use of the semantic package with other standard Emacs features are also part of this work.

Visually Manipulating Substructures Emacs offers at least three "outline" modes. These modes are used to show and hide various substructures of a document, like an outline. We have designed an outline minor mode for the PVS UI.

To hide a construct, we must have byte-accurate information about the source. This means that the parse tree must be up-to-date, at least with respect

to the subregion that is being hidden. As parsing does not take a negligible amount of time, we selectively update the parse tree on demand.

Typically a file is re-parsed only when its buffer is saved. To determine if the parse tree needs updating during outline operations, we first examine whether a parse of a file is necessary by examining the `buffer-modified` flag of its associated buffer. If the buffer has been modified, we attempt to re-parse only the region of the buffer that approximately corresponds to the construct being hidden. If this parse fails, a full file parse is initiated.

We also use "electric" actions to hint at when a construct has been significantly changed. For example, if a bracketing keyword like `THEORY` or `END` is added or deleted, it is likely the case that the parse tree needs to be completely updated. Likewise, large scale kills or yanks also trigger a full re-parse.

An idle-timer is used to trigger background parse tree updates when the user has not interacted with Emacs for a user-specified amount of time (typically around 30 seconds, much like the *lazy-lock* package's `lazy-lock-stealth-time`).

Structures are hidden and shown via modified mouse actions or via the standard keymaps of outline mode. A hidden structure is indicated with an elided hypertext region (automatically highlights when touched by the mouse pointer) containing four periods. This representation was chosen so that it is similar to the elision that PVS pretty-printing performs today, but different enough that it is clear the elision is due to outline mode and not PVS itself. Touching a hidden structure with the mouse pointer for an extended period of time will temporarily un-hide the structure for examination.

As with normal outline mode, incremental searches on a buffer with hidden regions finds matches in hidden text, and such matches are made temporarily visible. If the user exits the search within such a temporarily hidden region, the text remains visible.

In general, PVS outline minor should be familiar to users who have used other outline modes and should not distract the user from his or her work in any way.

Extended Implicit Browsing The final piece of new functionality added to the PVS UI is extended implicit browsing.

The Hyperbole package was the first environment for Emacs to provide extensive implicit hyperlinking [22]. A system is *implicitly hyperlinked* if visual constructs can be selected in some manner without the use of any explicit (e.g., literal hyperlinks) source text link constructs.

For example, automated spell-checking functionality, like that found in Emacs or Microsoft Word, is an example of implicit hyperlinking. In such an environment, the source document, perhaps a research paper, is implicitly hyperlinked to one or more dictionaries. The purpose of this particular hyperlinking is typically word spelling correction and definition and synonym lookup.

In general, the actions of an implicit hyperlink can either be informational (e.g., the definition of an operator or variable) or corrective (e.g., correcting the spelling of a variable).

In Hyperbole, an implicit hyperlink is activated by using what is known as the "action button". The default action button is a shifted middle mouse button, though of course the definition is user configurable.

Prior to this new work, PVS supported several kinds of implicit hyperlinks. Coincidentally perhaps, PVS supports the same default action, a shifted middle mouse button click. Using the action button, the type declaration of a construct is shown in a small information window. The key sequence `Control-.` (bound to the function `show-expanded-form`) is used to show the expanded form of a construct, and `ESC Control-.` (`goto-declaration`) is used to jump to the definition of a construct. Likewise, a set of `ESC`-prefixed commands let a user list all declarations in a theory, find where an identifier or declaration is used, etc. (See the various `*-declaration*` functions in the file `pvs-browser.el`.)

We improve on this functionality by increasing the number of recognized implicit hyperlink types. PVS keywords can now be activated to show their full definition related and usage information. Also, a variety of implicit link types can now be used in PVS comments including URLs, ISBN numbers, embedded image references, RFC titles, Emacs Info nodes, Texinfo cross-references, mail addresses, various compiler messages, pathnames, outline nodes, man pages, key sequences, table of contents entry, tag location, bibliography references, and some other more esoteric types[7].

We think that the types of particular utility for PVS are URLs, embedded images references, Info nodes, and key sequences. URLs are obviously useful to cite relevant papers or other source material for the documented construct. Embedded images can be used to show pretty-printed versions of specific terms, sequents, or proof structures, a la the "preview-latex" package[8]. Info nodes can be used to cross-reference other Emacs and PVS documentation, in particular the PVS release notes, which are the only part of the PVS documentation that at this time uses Texinfo. Finally, embedded key sequences (e.g., terms like "{ `ESC Control-.` }" which execute the specified key sequence when action selected) can be used to document PVS UI behavior and trigger PVS actions so that, for example, the PVS tutorial can be more interactive and automated.

We are sure that PVS users will find other surprising and interesting ways to use this new functionality.

5 Pretty-Printing in PVS

We have also added a user-extensible, type-aware pretty-printer to the PVS UI. This work is currently orthogonal to the semantic package-based work of Section 4.

The PVS input syntax is currently limited to ASCII characters. This is fairly restrictive, but PVS allows operators to be overloaded, using the type system

[7] See the file `hibtypes.el` in the Hyperbole distribution for more details.

[8] We are considering integrating the functionality of preview-latex into PVS but have not yet begun this work [8].

and theory hierarchy to determine the operator in any given context. Overloading is very convenient, and heavily used in mathematics; the context usually makes clear which operator is meant. For example, the PVS prelude has four declarations for the caret operator "^".

Pretty-printing the concrete PVS syntax is not difficult. Pretty-printing in PVS is handled using the Common Lisp Pretty Printing facility [19, Chapter 27] with a set of methods that walk down the PVS abstract term structures, which are implemented in CLOS.

In pretty-printing for LaTeX, with X-Symbol, UNICODE or other output, where there are many more operator symbols available, it may be desirable to map these PVS operators to output operators of a different format. For example, one might wish to use e^2 for `e^2` (exponentiation), and B_m^n for `B^(m, n)` (bit vector extraction). However, to properly translate such forms, more information is needed (in particular, types and resolutions).

The PVS LaTeX printer is driven from typechecked specifications. It allows substitutions to be made, and, by using the resolution information, can distinguish operators based on arity and the theory where the operator is declared. Types are not currently used, in order to keep the substitutions file simple[9].

Other approaches are available for pretty-printing that are especially useful when embedding different logics in PVS. For example, Skakkebaek [18] generated a new parser for the Duration Calculus, mapping its abstract syntax to subclasses of the PVS abstract syntax. Instances of these classes could be pretty-printed using specialized methods, but typechecking, proving, etc. would use the methods for the superclass. This works well, but it is not easy to define a new grammar in PVS[10]. Defining a new grammar for the Duration Calculus was not too difficult because it copied most of the PVS grammar and simply added a few new operators. Another difficulty is that as a proof is developed, terms get introduced that are part PVS and part Duration Calculus, and though they would be pretty-printed, the result could be confusing to the user.

In Pombo [17], PVS was used to provide the semantics of $\mathbf{A_g}$ specifications, defining the semantics of First Order Dynamic Logic and Fork Algebras, along with rules and strategies that allow a user to reason in $\mathbf{A_g}$. Here there were conversions defined, such as a meaning function, and arguments such as the current world of the Kripke structure, that by default are included in the prover interaction, but add clutter to the proof. In this case the function for pretty-printing applications was modified in order to suppress the meaning function and the world argument.

A better approach would be to define a "hook" that allows a user to specify how to print specific terms. This would be similar to the hooks used in Emacs, which are lists of functions that are invoked one after another until one of them succeeds. We expect to add such a hook to the PVS pretty-printer that lets a user

[9] In order to properly provide support for types, parts of the substitutions file would need to be typechecked, and it is not trivial to determine the typechecking context.

[10] PVS currently uses the Ergo Parser Generator [10], which has a number of quirks that make it difficult to use.

register pretty-print functions on a per-type basis. Each hook function would check if the term given is one that requires special pretty-printing treatment, and would then perform the pretty-printing and return an indication that the pretty-printing has been done. If none of the hooks are applicable, then the default pretty-printing is used. We expect that some general functions will be provided that make it easy to perform common tasks, like suppress arguments, and recognize applications whose operator is a constant of a specified theory and type. In its full generality, these hooks act as semantic attachments for the customized pretty-printing of terms of any form.

This works fine for some things, but not all pretty-printing tools can be integrated into Common Lisp. For such tools the resolution and type information should be provided. The most promising approach to this is to provide the abstract syntax in XML because most existing languages have facilities for reading XML documents. This is done, for example, in OMDOC, which provides an extension for PVS that generates OMDOC abstract syntax from typechecked PVS specs [9]. There are future plans for generating XML that directly reflect the internal abstract syntax of PVS, and this could easily be used to build a new pretty-printer.

5.1 X-Symbol Integration

Even though there are limitations in pretty-printing heavily overloaded expressions in PVS, we have integrated initial support for interactive pretty-printing using the X-Symbol Emacs package [21].

X-Symbol is a package capable of interactively rendering specific character sequences in a buffer with non-ASCII characters (e.g., actual mathematical symbols, Greek letters, etc.). A standard set of mappings for all the core PVS operators is provided with our new PVS UI extensions. All uses of an overloaded operator look identical at this point in time.

While writing PVS specifications and performing proofs in PVS, cutting and pasting terms is a very frequent user operation. Unfortunately, when a character sequence rewritten with X-Symbol is copied in Emacs (whether through a mouse or keyboard action), exactly the *rendered* text is put into the copy buffer. The original source text is lost, and it is what is needed for a legitimate paste, since PVS does not understand the pretty-printed characters.

We have circumvented this cut-and-paste problem in the following fashion. In the margin of the pretty-printed buffer adjacent to every pretty-printed construct is a commented elided region. The region is just a comment from PVS's point of view, and Emacs is not rendering the contents as it is hidden via the PVS outline minor mode. Inside of this comment is the original source text of the pretty-printed construct. Therefore, when a user needs to cut-and-paste a term, she must only double-click on the elided region to obtain a copy of the term in question.

We find this mode of interaction quite convenient but believe that more experience with mixed-mode representations in theorem proving environments is necessary.

6 Conclusion

There remain several new features and refinements that we would like to make to the new PVS UI.

First, we have not yet added a graphical toolbar to represent some of the new features. We have witnessed that some Emacs users will simply ignore new features until they have such toolbars, so we plan on adding them soon.

Keeping track of the importing and exporting relationships between PVS theories is a complex and sometimes time-consuming task. A new semantic package that is under development called COGRE can graphically represent semantic package parsed structures [11] (initially, COGRE's focus is UML diagrams). In COGRE, a graphical representation can be manipulated to change the underlying associated data. We think some experimentation with the graphical representation of theory relationships is warrented.

Finally, UI operations involving parameterized theories are weak. We would like to better support completion and summarization of such parameterized substructures.

This work will be made available in late 2003, probably as part of a new PVS 3 distribution.

Acknowledgments. This work was supported by the Netherlands Organization for Scientific Research (NWO). Thanks to Erik Poll, Martjin Warnier, Bart Jacobs, and Adriaan de Groot for their input on this paper.

References

1. Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating maple and pvs. In *Theorem Proving in Higher Order Logics, TPHOLs 2001*.
2. David Aspinall. Proof General: A generic tool for proof development. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *Lecture Notes in Computer Science*. Springer–Verlag, 2000.
3. Klaus Berndl, Jesper Nordenberg, Kevin A. Burton, and Eric M. Ludlam. The ECB user manual, July 2003. See http://ecb.sourceforge.net/ for more information.
4. Janet Bertot and Yves Bertot. CtCoq: A system presentation. In *Algebraic Methodology and Software Technology*, pages 600–603, 1996.
5. R. Bornat and B. Sufrin. Jape's quiet interface. In *Proceedings of User Interfaces for Theorem Provers (UITP'96)*, 1996.
6. The GNU Foundation. The GNU bison manual, February 2002. See http://www.gnu.org/software/bison/bison.html for more information.
7. Inc. JetBrains. IntelliJ IDEA 3.0 overview, 2002. See http://www.intellij.com/ for more information.
8. David Kastrup et al. The Emacs preview-latex package, 2003. See http://preview-latex.sourceforge.net/ for more information.
9. Michael Kohlhase and Sam Owre. An OMDoc interface to PVS, 2001. See http://www.mathweb.org/cvsweb/cvsweb.cgi/omdoc/projects/pvs/ for more information.

10. P. Lee, F. Pfenning, J. Reynolds, G. Rollins, and D. Scott. Research on semantically based program-design environments: The Ergo project in 1988. Technical Report CMU-CS-88-118, Department of Computer Science, Carnegie Mellon University, 1988.
11. Eric Ludlam. The COGRE manual, 2002. See http://cedet.sourceforge.net/cogre.shtml for more information.
12. Eric Ludlam. The Semantic manual, 2002. See http://cedet.sourceforge.net/semantic.shtml for more information.
13. Eric Ludlam et al. The Speedbar manual, 2003. See http://cedet.sourceforge.net/speedbar.shtml for more information.
14. Xref-Tech Marian Vittek. The Xrefactory system, 2002. See http://www.xref-tech.com/ for more information.
15. N. Merriam and M. Harrison. What is wrong with GUIs for theorem provers? In *Proceedings of User Interfaces for Theorem Provers (UITP'97)*, 1997.
16. Sam Owre et al. The PVS 3.0 ChangeLog, 2003. See http://pvs.csl.sri.com/ for more information.
17. Carlos López Pombo, Sam Owre, and Natarajan Shankar. A semantic embedding of the $\mathbf{A_g}$ dynamic logic in *PVS*. Technical Report SRI-CSL-02-04, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, July 2003. To appear.
18. Jens U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, December 1993.
19. Guy Steele. *Common Lisp: The Language*. Digital Press, second edition, 1990.
20. Guido van Rossum et al. The Python language documentation, 2002. See http://www.python.org/ for more information.
21. Christoph Wedler. The X-Symbol manual, May 2003. See http://x-symbol.sourceforge.net/ for more information.
22. Bob Weiner et al. BeOpen.com Hyperbole: The everyday net-centric information manager, July 1999. See http://sourceforge.net/projects/hyperbole/ for more information.

Proving as Programming with DrHOL: A Preliminary Design

Scott Owens and Konrad Slind

School of Computing, University of Utah

Abstract. We discuss the design of a new implementation of the HOL system aimed at improved graphical user interface support for formal proof. We call our approach *Proving as Programming*, since we believe that metalanguage programming is a central aspect of proof construction. Thus we look to contemporary programming environments for inspiration on how to provide graphical support for proof. In particular, our implementation builds upon *DrScheme*, a popular programming environment for Scheme.

1 Proving as Programming

We have begun work on *DrHOL*, a new implementation of the HOL logic. DrHOL is systematically derived from HOL-4 [8] and aims at improving user interfaces in many aspects of work in HOL: development of proof procedures, construction of terms and definitions, interactive proof, and embedding of object languages are seen as candidates for better interface support. We believe that programmability is an essential part of all these activities. To support our view, we will discuss the ways in which we will adapt an advanced programming environment, DrScheme [3], into a proof environment for HOL-4. The main question being investigated—and it will take a while to obtain comprehensive answers—is : *How can support for programming also support proof*?

The issue of programmability in proof is interesting. One one hand, there is a long history of tactic proofs, stemming from the invention of tactics and tacticals in the original LCF system. Since tactics and tacticals are metalanguage programs, the construction of compound tactics to prove a goal can be considered programming. Thus we have a large body of historical evidence that programmability is *A Good Thing*. On the other hand, the *declarative* approach to proof has recently garnered much attention[1] [10, 6, 13, 9, 11], and is often presented as a way of abolishing programming from the activity of constructing proofs. A declarative proof is written in a fixed proof language, which may be processed in a variety of ways. Importantly, the specification of a proof in the proof language is independent of the means of achieving the proof. This allows declarative proofs to be processed in more ways than tactics, which can only be executed. As a result, declarative proofs are more readable and maintainable than tactic proofs. However, declarative proofs can be more verbose than

[1] Although its roots in the Mizar system are quite old by the standards of the field.

procedural proofs, and may contain more explicitly given terms. That the two approaches are not completely distinct is illustrated in [12], which demonstrates that declarative proof may be quite concisely implemented via tactics.

One of the early reasons for having programmability was *extensibility*. Since most of the basic tactics for LCF embodied quite small reasoning steps, composing them into larger steps was accomplished by tacticals and, often, by extended ML programming. Later work, especially in Isabelle, showed that long and intricate tactics could often be replaced by parameterized proof tools, such as first order provers and simplifiers. The amazing increases in computational power available to researchers have validated the move to more powerful proof tools. Interactive theorem proving has thus become much more efficient in terms of user time and effort, with the obvious *caveat* that when one gets well and truly stuck on a proof, no amount of automation will help.

However, in extended proof developments, we still find that highly automated tools are not completely adequate: the ability to write *ad hoc* proof procedures and custom term construction functions on the fly is a key facility. Similarly, when a sequence of steps re-occurs in proof, then programming is needed to avoid unnecessary repetition: this is just code reuse by procedural abstraction. To us, this implies that there is no real gap between online and offline construction of proof procedures. Thus a proof environment should support interactive construction of programs.

Proving as programming should not be confused with *Proofs as Programs*, a slogan associated with constructive logic. That slogan has specific and deep technical content behind it, while our slogan is more a methodological attitude.

2 DrScheme

DrScheme is a graphical program development environment for the Scheme programming language. DrScheme presents a single window with two nested windows, called the *definitions window* and the *interactions window*. The definitions window (the upper one) contains a program that users can execute, save to disk, and access in the interactions window. The interactions window (the lower one) provides a "read-eval-print loop" (REPL), in which users can experiment with their programs. Both windows implement the same programming language, use the same error messages, and report result values using the same syntax. When a user presses the "execute" button the interactions window is cleared, the program in the definitions window is executed and the resulting definitions are placed in the interactions window for the programmer to inspect and manipulate. The programmer can handle infinite loops with the "break" button. When pressed, the break button stops the currently executing program within either window. DrScheme provides the usual emacs-like program editing features in both the definitions and interactions window, such as auto-indenting and parenthesis matching. The definitions window has a *Check Syntax* feature that draws graphical arrows from a variable use to its binding when the user moves the mouse over the variable. Check Syntax also implements α-consistent variable

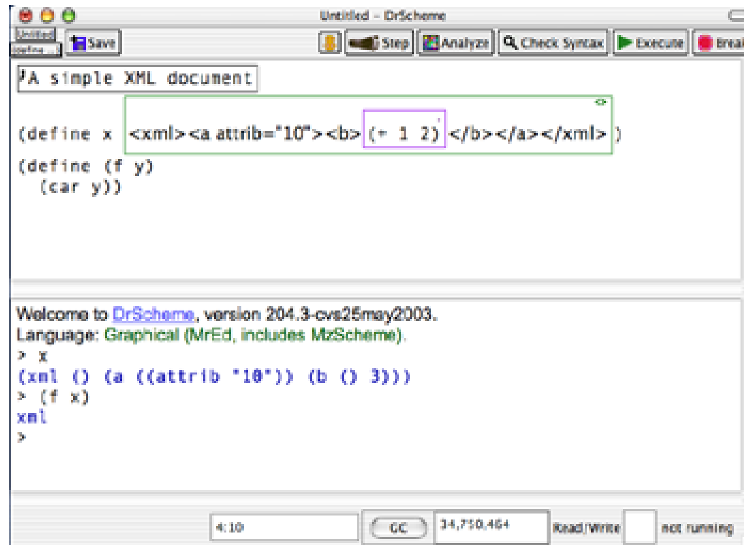


Fig. 1. DrScheme

renaming. While most program editors can graphically highlight the locations of compilation errors in the program's source text, DrScheme can also highlight the locations of runtime errors and uncaught exceptions when they occur.

To support both student and professional programmers, DrScheme provides several different language levels. A *Language Level* is a particular dialect of Scheme tailored to the needs of a certain class of students or professionals. Students usually work with a subset of Scheme that they can completely understand. This enables DrScheme to provide error messages that mention only concepts the students already understand. For example, the "Beginning Student" language level does not have anonymous functions and syntactically restricts the function position of an application expression to be a variable. Students often have difficulty at first in placing parentheses correctly and might write ((+ 1 2)). A Scheme system would report a runtime error indicating 3 is not a function. However, Beginner Scheme reports a compile-time error indicating that the expression (+ 1 2) appears in the function position of ((+ 1 2)) where only a variable is allowed.

DrScheme is widely used to teach beginning programmers; the language levels are used to structure the course and have been proven and classroom tested over the past 8 years. The text [2] serves as the foundation of the course.

Each language level is implemented by a compiler that translates programs from the language level's dialect into DrScheme's primitively supported Scheme dialect. For student language levels, these compilers primarily detect static errors and introduce code for better explanation of run-time errors.

Besides levels of a particular language, DrScheme can also support different languages: a proof-of-concept ALGOL60 language level exists, and work is underway to support several Java and OCaml language levels for pedagogic purposes [5]. Although these language implementations are technically similar to levels within a language (each being basically a compiler from the particular language to Scheme), they operate on a much grander scale. In particular, care must be taken to ensure that these languages interoperate nicely with Scheme programs.

The core of the DrScheme system is an interpreter called *mred* (pronounced 'Mister Ed'). mred interprets a dialect of Scheme that includes integrated graphical interface widgets. DrScheme itself is just a program written in this dialect that executes on the mred interpreter. DrScheme executes the user's programs directly in the mred interpreter, which supports sophisticated primitives that allow DrScheme to work robustly in the presence of misbehaving user programs [4]. Higher language levels can use the same facilities. DrScheme can run on the platforms that mred supports: Apple, Microsoft Windows and Unix with X-windows.

3 DrHOL

The HOL theorem prover is implemented in the SML programming language as a library of SML functions. The two main activities in HOL are constructing new logical terms and performing inference steps. Both of these activities require programming in SML. In the case of terms, the proper SML data constructors must be invoked to generate the term[2] and in the case of inference steps, the SML function that performs the inference step must be invoked. Occasionally a new inference rule or tactic needs to be written in terms of other existing inference functions. These facts lead us to the central conclusion that theorem proving in HOL is an inseparable activity from programming in SML. Hence our thesis that theorem proving in HOL should occur in a programming environment.

DrHOL is being implemented as a language level in DrScheme with a two-step approach.

1. An SML language level will be implemented via an SML to Scheme compiler. This will allow the existing body of HOL source code to be run inside the DrScheme environment and to interoperate with Scheme programs. Thus a HOL user will be able to use HOL in the DrScheme environment as he does currently. However, he will have access to the helpful programming features of DrScheme.
2. HOL itself will be extended and modified to support greater integration into DrScheme. This step will allow more advanced user-interface features to be added to HOL, while maintaining a tight integration with the development environment.

[2] Built in custom term parsers simplify this task

3.1 SML to Scheme

We have chosen to provide programming environment support for HOL by creating a general purpose SML to Scheme compiler, although this is by no means the only possible approach. Because both HOL and DrScheme will be Scheme programs running on the mred Scheme interpreter, our approach will support a tight integration between the theorem prover and programming environment. Our SML to Scheme translation maps each externally visible SML construct into an equivalent Scheme construct, so that interoperation between Scheme and SML programs will not be difficult. For some SML constructs, such as structures, Scheme has no suitable built in construct. However, using Scheme's macro system, we can create new Scheme constructs without exposing implementation details to the programmer.

To simplify the task of building an SML compiler, we use the parser, overloading resolver and type checker from the Moscow ML compiler, all of which are written in SML. We currently invoke this front end in a separate process, but eventually intend to bootstrap the front end by translating these parts of the Moscow ML compiler into Scheme with our compiler.

We will briefly discuss several approaches that avoid building an SML to Scheme compiler and explain why we have not chosen them.

- *Use an SML interface widget package to build DrHol in SML* [7].
  Building a new extensible, production quality programming environment is a much more difficult and time consuming project than building an SML to Scheme compiler. Moreover, MoscowML, our current ML platform, doesn't support threads.
- *Translate HOL into a Scheme program manually.*
  While this might be feasible for some small core of HOL, the entire system is altogether too large. Furthermore, it's not *future-proof*: we want to be able to use future HOL libraries that will be written in SML.
- *Invoke a separate Moscow ML process from DrScheme to evaluate SML programs.*
  Two-process systems have been tried and found, in our opinion, to be less robust than desired. They have problems with dealing with undesired behaviour (breaking loops, interpreting error messages) and with interpreting returned values.

3.2 Safety

Via the SML type system, HOL guarantees that any theorem in the system has been proven through the application of a series of basic inference steps to some basic theorems. Knowing that theorems have been built using only a few simple and well-tested rules provides a HOL user with confidence that theorems produced in HOL are indeed correct. It is quite important that we provide this invariant in our Scheme system, not only when dealing with translated SML programs, but also when dealing with Scheme programs that use parts of the HOL

system. Thus our compiler must preserve the observational equivalence relation for SML programs when put into Scheme contexts as well as SML contexts.

HOL implements theorems as an SML datatype and relies on the type system to ensure that no data constructed otherwise can be considered as a theorem. We ensure this property holds even when Scheme programs handle theorems by translating SML's datatype constructors into mred's *define-struct* form, which provides data abstraction facilities. HOL uses SML's structure system to restrict access to the theorem constructors, so that only the basic inference steps may see them. Since we translate SML structures into a Scheme implementation of structures, the theorem constructors are protected from Scheme programs just as they are protected from SML programs.

4 Benefits of DrHOL

HOL users will benefit from the integration of HOL into DrScheme almost immediately by taking advantage of the programming features of DrScheme listed previously. Beyond that, DrHOL will be extended to support more advanced capabilities ranging from modest extensions to ambitious projects. In the end, we hope our system will provide a better system for interacting with HOL than current alternatives.

4.1 Definitions and Interactions

Although [1] shows how HOL may be accessed purely through a language-neutral API, currently the most common way to interact with HOL is through an SML REPL. For many tasks, REPL interaction is ideal. Users can quickly try many different approaches to proving a theorem and get immediate responses from the system. DrScheme's interactions window provides a convient-to-use REPL whose robustness surpasses most existing interactive modes for emacs.

Once exploration is finished the user needs a record of construction of the expression (be it a type, term, definition, theorem, or more complex entity). The record needs to be easily executed on demand, because the expression might be needed in the construction of another expression. Currently users must carefully look back through their interactions buffer and find the steps that made actually progress in the construction and manually copy them into a separate file. We will be able to assist the user in this process by providing automatic support for moving expressions from the interactions window into the definitions window.

4.2 Graphical Syntax

DrScheme supports the encapsulation of syntax in graphical containers. When a programmer adds a graphical container to the interactions or definitions window, the container is treated as a single character by the editor while the cursor is outside of the container. When the cursor is placed inside the container the programmer can edit the container's contents as though it were a separate window.

These containers can be used to implement different syntaxes in much the same way as Moscow ML's antiquote or Lisp's quasiquote and unquote mechanisms. Compared to these techniques, graphical containers for syntax provide a much more easily recognized visual cue to the programmer that a different syntax is in use. Furthermore, none of the container's contents need to be prefixed with escape sequences since its extent is delimited graphically instead of textually.

DrScheme currently supports two different kind of containers: comment boxes and XML [3] boxes. The comment boxes contain comments whose contents are ignored. XML boxes contain literal XML text typed in directly. The contents of an XML box are converted to an s-expression when the box is encountered by the parser. The programmer can also insert a Scheme box into an XML box. The Scheme box's contents are evaluated as a Scheme program and the results placed into the XML box's resultant s-expression. The presence of the XML box lets DrScheme know exactly how the contents of the box should be treated. Inside an XML box, DrScheme automatically inserts the matching closing tag for each opening markup tag the programmer writes.

DrHol will provide HOL term boxes to allow the programmer to input and output HOL terms directly and in a natural syntax. For example, the HOL term box can automatically replace LaTeX like symbol commands (`\alpha`) with the actual symbol. It will also be able to color the term syntax to distinguish between logical constants, free variables and bound variables and, via Check Syntax, provide graphical arrows linking their definitions and uses. Interoperability with external tools may be achieved via exporting types, terms, theorems, and theories in emerging standard XML-based formats such as OpenMath or OMDoc.

4.3 Help System

We will be able to integrate the existing HOL help system with the programming environment. For example, the user will be able to select a HOL function with the mouse and open a new window with the documentation for that function. The user will also be able to select a logical constant from a HOL term and retrieve its definition. These facilities are more extensive and easier to use than the current HOL help system which either (a) dumps out a textual message to the interaction loop, thus obscuring the proof state; or (b) depends on an external web-browser.

4.4 Goal Stack Management

HOL keeps a global state that tracks which obligations remain in proving some particular theorem. These obligations naturally form a tree, however for historical reasons HOL uses a *goalstack* interface that tracks only the leaves of the tree. Although this is one of the most heavily used tools in HOL, DrHOL will provide user-interface support for managing proof state directly as a tree. We expect that these additions will require moving beyond the two-window format

[3] XML is the W3C's eXtensible Markup Language.

that DrScheme currently supports, with an extra window that directly displays either the proof tree or goal stack. Then the user can graphically navigate the remaining proof obligations and DrHOL can automatically generate the final proof script from a completed proof tree.

4.5 Embedding other logics in HOL

A common activity in HOL is to embed computer languages and logics. This approach allows the user of the domain-specific logic to use all of HOL's tools and power when proving things in the logic as well as when constructing proof tools for the logic. Sometimes, however, the embedded language doesn't require such an extensive and complicated interface. Occasionally, the logic's only interface should be graphical. Our system would allow a domain-specific GUI extension to the DrHOL environment to be constructed alongside the logic's embedding. In particular, an embedding may require support for different syntax or even point-and-click proof tools.

5 Future Work

Although a future work section may seem strange in a design paper, we can see goals that will be achievable once our basic design has been implemented.

- We can implement a language level that supports only a subset of the functionality of the HOL system. For example, one could envision a language level that introduced 'the essence of HOL' by restricting inference steps to only natural deduction style introduction and elimination rules for the logical connectives (as was done in Tom Melham's successful HOL course). Another language level useful for teaching beginners would be tactic-based, along the line of a *Ten Tactic HOL* method of teaching beginners (this teaching approach was used by Graham Birtwistle in the late 1980's). In such a level, a goal stack window would have a fixed number of buttons corresponding to the fixed repertoire of allowed tactics. Such restricted environments would be able to offer improved support for beginning users, by thoroughly supporting a comprehensible collection of inference tools.
- With an integrated theorem prover available in the programming environment, Scheme or SML programmers could have ready access to theorem proving for developing and analyzing their programs. To take a simple but challenging example, one could envision a language level which enforced termination, via HOL proof, of each recursive function introduced by a user.

6 Conclusion

We have presented a preliminary design for an SML language level in DrScheme, and on top of that, a level for the HOL-4 implementation of the HOL logic. This system, DrHOL, will provide an extensible basis for integrating and investigating

user-interface support for interactive theorem proving. Underlying our design is the assumption that 'if proving is programming, then a good programming environment can be adapted to be a good theorem proving environment'. To give some substance to this viewpoint, we have discussed specific areas where the facilities of DrScheme can be used to improve proof development. We expect that many other such opportunities will arise as this work matures.

A major source of encouragement for us is the vigorous ongoing development of DrScheme by its talented group of developers. Just as we plan to adapt their insights about graphical support for programming, we hope our work on support for proof in DrHOL will provide useful ideas and challenges in the future development of DrScheme.

References

1. L. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham, *The PROSPER toolkit*, STTT: International Journal on Software Tools for Technology Transfer **4** (2003), no. 2, 189–210.
2. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, *How to design programs*, The MIT Press, Cambridge, Massachusetts, 2001, `http://www.htdp.org/`.
3. Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen, *DrScheme: A programming environment for Scheme*, Journal of Functional Programming **12** (2002), no. 2, 159–182, A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.
4. Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen, *Programming languages as operating systems (*or *revenge of the son of the Lisp machine)*, Proc. ACM International Conference on Functional Programming, September 1999, pp. 138–147.
5. Kathryn E. Gray and Matthew Flatt, *ProfessorJ: A gradual intro to Java through language levels*, OOPSLA Educators' Symposium, October 2003.
6. John Harrison, *A Mizar mode for HOL*, Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96 (Turku, Finland), Lecture Notes in Computer Science, no. 1125, Springer-Verlag, 1996, pp. 203–220.
7. C. Lüth and B. Wolff, *Functional design and implementation of graphical user interfaces for theorem provers*, Journal of Functional Programming **9** (1999), no. 2, 167– 189.
8. M. Norrish and K. Slind, *A thread of HOL development*, The Computer Journal **45** (2002), no. 1, 37–45.
9. Donald Syme, *Three tactic theorem proving*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 203–220.
10. M. Wenzel and F. Wiedijk, *A comparison of the mathematical proof languages Mizar and Isar*, Journal of Automated Reasoning **29** (2002), 389–411.
11. Markus Wenzel, *Isar—a generic interpretative approach to readable formal proof documents*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 167–185.

12. Freek Wiedijk, *Mizar Light for HOL Light*, Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001 (Edinburgh), Lecture Notes in Computer Science, no. 2152, Springer-Verlag, 2001, pp. 378–393.
13. Vincent Zammit, *On the implementation of an extensible declarative proof language*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 185–202.

User Interface for Adaptive Suggestions for Interactive Proof

Martin Pollet Erica Melis Andreas Meier

Fachbereich Informatik, Universität des Saarlandes,
and DFKI Saarbrücken,
66041 Saarbrücken, Germany
email:{pollet,melis,ameier}@ags.uni-sb.de

Abstract. We describe an interaction console for interactive proof exercises using proof planning as back-engine. Based on the proof planning situation, the console offers suggestions for proof steps. These suggestions can dynamically be adapted, e.g. to the user and to pedagogical criteria. This adaptive configuration is possible because the suggestion mechanism and proof planning are clearly separated. Therefore, a configuration can represent pedagogical knowledge which would not be possible by using proof planning alone.

1 Motivation

Our motivation for the described work and its follow-ups is the improvement and system-support of learning to prove mathematical conjectures. Empirical studies [11] suggest that students' deficiencies in their mathematical competence with respect to understanding and generating proofs are connected with the shortcoming of students' self-guided explorative learning opportunities and the lack of (self-)explanations during problem solving. Such explorative learning can be supported by tools. This motivated the integration of the interactive proof tool ΩMEGA into the user-adaptive learning environment ACTIVEMATH [7]. Moreover, empirical results suggest that instruction with proof planning methods can be a learning approach that is superior to the traditional learning for proof [8]. This motivates the implementation and integration of the interactive proof planning component of ΩMEGA into ACTIVEMATH. However, integrating any interactive system is insufficient for education because for effective learning more requirements have to be met.

What is necessary in order to support learning mathematical proof?
In general, an educational context requires additional features (compared with a mere assisting system) for effective learning such as

- adaptivity [5] and
- faulty proof attempts whose (coached) discovery and repair are a major source of learning [13].

Adaptation and personalization may effect the content, the possible or preferred problem solving alternatives/strategies, the level of detail of proof, the appearance, etc. For instance, an exercise about the limit of a concrete sequence can be solved by either using only the definition of the limit or by applications of theorems. The choice for one of the proof ideas should depend on the learner's capabilities and knowledge.

Erroneous situations or faulty suggestions are not appropriate for a system that is devised for problem solving assistance. For a learning tool, however, it might not be the best idea to always make only correct suggestions as it is for an assistant system because then the student might just click on suggestions rather than learn anything. For instance, when the student should prove the limit of a sequence by applying theorems and she is already familiar with the theorems then it would be to restrictive to suggest only applicable theorems.

The decision on when to make which faulty suggestions will, of course, depend on the student's situation and on her capabilities. In more detail, it depends on the student's learning goal, the learning context, her learning history, and the competency of the student (all represented in ActiveMath's student model) as well as on the pedagogical strategy. Therefore, the suggestions should be dynamically generated depending on the student model and on the pedagogical strategy.

The main contribution of the paper is the description of the technology underlying the adaptable suggestions, adapted problem solving strategies, and their usage. This includes the representation of configurations comprising the proof strategy (proof idea) and a set of agents each representing reasons for suggesting a particular proof planning method in the proof process. This also includes the interaction and suggestion mechanism as well as the adaptation functionality. It shows how agents can be tailored to serve certain mathematical criteria or learning goals that are represented in ActiveMath's student model. It describes how the student can employ different proof planning strategies and take advantage of the proof planner's automatic mode that is running in the background while the student interactively attempts to prove a theorem.

2 Preliminaries

2.1 ActiveMath

ActiveMath is a web-based learning environment (for mathematics) [7]. It generically integrates several back-engines for exercising and exploratory learning – among them the computer algebra systems Maple and MuPad and the theorem proving system Ωmega.

One reason for using proof planning in a learning environment for mathematics is that methods represent typical steps in mathematical reasoning. Therefore, the communication and teaching of those methods should enable a student to extend and improve their reasoning skills. There is some empirical evidence for this hypothesis [8].

ActiveMath generates (mathematics) learning material user-adaptively, i.e., dependent on the learner's goals, learning scenarios, preferences, and mastery level. The adaptivity is based on a *student model* that includes

- the history of the learner's actions
- her preferences
- her mastery level of concepts and skills with a range between 0 and 1.

The student model is updated by diagnoses of the learning activities such as reading and problem solving. A student's exercise performance is evaluated and the evaluation is passed to the student model for updating it.

This student model can be queried by the mechanisms cooperating with Ωmega. For instance, the suggestion mechanism may query the history in how many previous exercises the student correctly applied the method M in an `only-correct` setting and what the student's mastery level of M is, in order to determine whether the next exercise will include an application of M in an `also-faulty` setting.

2.2 Ωmega

For interactive proof exercises two functionalities of the theorem proving system Ωmega [12] are employed: automatic proof planning [9] and the support of interactive proof planning by an agent-based command suggestion mechanism [2]. The combination of proof planning and the suggestion mechanism is described in the Section 3, but first we introduce them separately.

Proof Planning. Proof planning in Ωmega was originally conceived for *automated* theorem proving. Proof planning is a technique in theorem proving that aims at reducing the search space by proving at the level of *methods* which can encapsulate complex proof steps and by introducing meta-level guidance [3, 10].

Proof planning starts with a goal that represents the conjecture to be proved. It continues by applying methods for which the application conditions are satisfied and this generates new assumptions or reduces a goal to subgoals until no goal is left. The resulting sequence of instantiated methods constitutes a solution proof plan.

Successful proof construction requires also to construct mathematical objects, i.e., to instantiate existentially quantified variables by witness terms. In proof planning meta-variables are used as place holders for witness terms, and the planning process proceeds until enough information is collected to instantiate the meta-variables.

The proof planner has an automatic and an interactive mode. In the automatic mode the proof planner searches for a solution proof plan, i.e., in each intermediate state it searches for applicable methods and valid instantiations. Mathematics-oriented heuristics guide this search. In interactive proof planning, the user makes the search decisions and these include the choice of methods and the instantiation of meta-variables.

To structure the available proof planning methods and make the proof planning process more hierarchical, strategies have been devised. A proof planning strategy is specified by a set of methods and search heuristics. Different proof planning strategies correspond to and implement different proof ideas.

Command Suggestion Mechanism. ΩMEGA has a mechanism to suggest applicable tactics and their parameters. A suggested command usually consist of the name of a tactic and a list of parameters, e.g., the proof lines which the tactics should be applied to. The mechanism is implemented by agents. Each tactic and each argument is represented by one agent which checks whether there are instantiations of the command in the current proof context. We reuse this mechanism in the context of interactive proof planning for the generation of method suggestions.

3 Interactive Exercises with ΩMEGA in ACTIVEMATH

Interactive exercises with ΩMEGA are represented by OMDocs [4]. This representation includes a textual and/or diagrammatic representation, a formalization of the theorem, and ΩMEGA-specific commands. When a student decides to perform an ΩMEGA-exercise during a session, ACTIVEMATH starts ΩMEGA with its graphical user interface and a (bi-directional) XML-RPC connection is established between ΩMEGA and a proxy in ACTIVEMATH. The formal description of the exercise and the commands are sent to ΩMEGA and the interactive proof planning dialog is launched. When the exercise is finished by the user, some information about the ΩMEGA-session is sent to ACTIVEMATH's user model. For instance, the number of successful and of faulty method applications for relevant methods is returned, the number of unjustified lines, and whether the automatic mode was used.

Previously, proof planning in ΩMEGA, in particular, its graphical user interface was not well-suited for interactive proof learning by students since it requires too much attention for and knowledge about the system handling and could not be adapted to the student's situation.

3.1 Interaction Console for Proof Planning

In order to improve ΩMEGA's education potential we reduced the complexity of the required interaction by developing the interaction console. This is a first step for improving pedagogical usability.

Fig. 1 shows ΩMEGA's user interface enriched by the interaction console in a stage, where methods have already been applied to the initial problem. In the interactive mode the user has to specify which goal to work on next, the method (and possibly parameters) she wants to apply, etc. In order to help the user to concentrate on the proof construction, the interaction is via a console only (shown in Fig. 2 in more detail). The relatively simple dialog window offers restricted suggestions rather than ΩMEGAs full functionality.

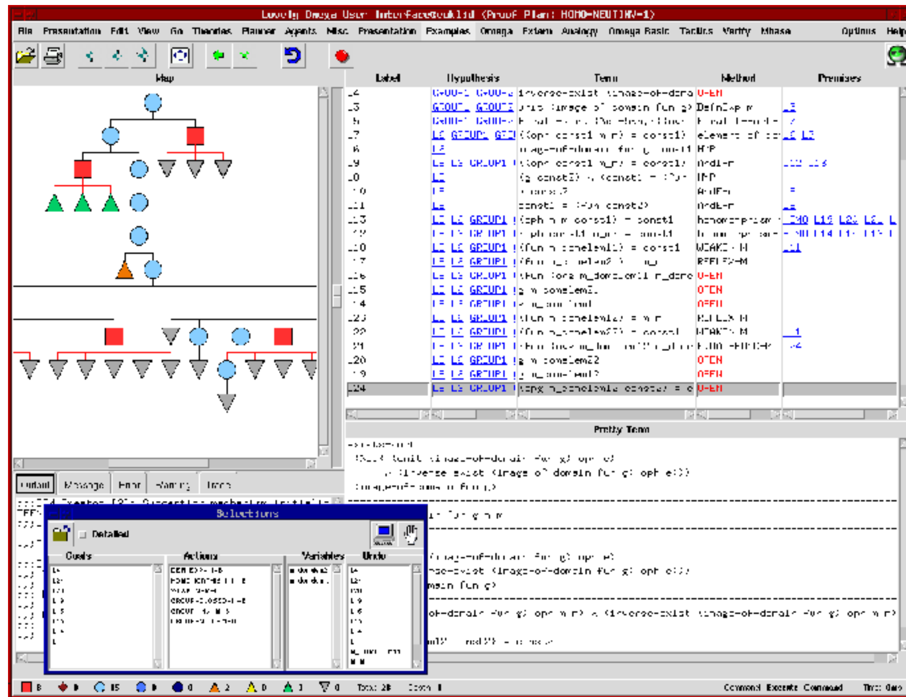


Fig. 1. The user interface with a graphical proof presentation, the sequence of proof lines, and the interaction console.

The user can choose a proof line (goal) for which several method names are suggested (actions). When the student selects an action, she may be asked for additional parameters, e.g., which proof lines should be used as premises. Then, the proof planner tries to apply the method. The next column of the interaction console (variables) displays the meta-variables of a proof plan for which a witness can be provided by the user. The last column allows the user to delete proof steps and undo instantiations of meta-variables.

During interactive proof planning the proof planner runs in the background, checks the applicability of methods selected by the user and provides feedback. The button with the computer symbol at the top of the interaction window starts the automatic proof search of the proof planner. The purpose is to help the user in case she got stuck altogether during the proof construction. Furthermore, the automatic mode of the proof planner can give hints on how to proceed. This functionality can be disabled by ActiveMath, if it prevents learning (see, e.g., [1]). The button with the hand interrupts the automatic mode and switches back to interactive proof construction. We also implemented a version in which the proof planner applies only one method in automatic mode.

To make the most frequent user interactions easy to perform only few key strokes should be necessary for those interactions. Therefore, selecting a method's

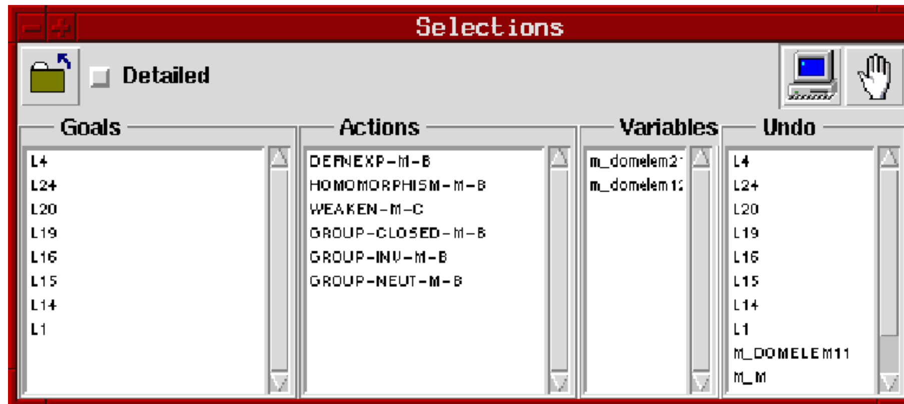


Fig. 2. The interaction console for interactive proof planning.

name for input is a better design than typing full method names. Moreover, to remember a name causes more working load than to recognize the name as an appropriate one. Therefore, again choosing a method from a list of suggested ones causes less distraction from the actual learning and understanding of proof.

3.2 Method Suggestions by Agents

The degree and kind of assistance can vary depending on the learning situation. For this purpose, the agent-based suggestion mechanism is used to create the suggestions of actions and their parameters. For each method one or several agents are implemented. Each agent analyses the proof situation and returns the method as a suggestion, if certain criteria specified in the agent-declaration are fulfilled.

Note that the suggestion agents are independent from the proof planner and from the application conditions of the methods. Agents can suggest methods that are not applicable at all, or can suggest parameters for a method which do not fulfill the application condition of the method. That is, depending on the agent's design the freedom to interact with the proof planner can be more or less restricted, more or less guided, and they can represent certain pedagogical strategies as described in section §3.3. In particular, faulty suggestions can deliberatively be introduced for learning from failure.

Agents for interactive proof planning are currently implemented for the following classes of problems:

- limit theorems [10],
- group homomorphisms,
- properties of residue classes [6].

3.3 Configurability and Adaptability

A configuration consists of a problem solving strategy and a set of agents, one agent for each method of the strategy. A configuration can encode pedagogical goals, ideas, or needs.

Choice of Strategy. There can be different proof planning strategies available for the configuration of an exercise. An example for selecting a strategy adaptively is the following. For proving problems about properties of residue classes there exist three different proof planning strategies. The strategies implement different ways to prove the theorems. The first strategy tries to apply theorems, the second strategy reduces the problem to an equation for which a general solution must be found, the last strategy introduces a case split over the (finitely many) elements of a residue class. The decision for a strategy can depend on the knowledge of a student (whether she knows the theorems that are the prerequisites of the first strategy) but can also result from her performance in previous exercises in which, e.g., the other strategies have been trained already. Such configuration heuristics can be encoded by pedagogical rules. For instance, put naturally

```
IF studentKnowledge(prerequisites (firstStrategy)) > medium
AND studentKnowledge(firstStrategy) < medium
THEN present exercise-for(firstStrategy)

IF  studentKnowledge(firstStrategy)  > medium
AND studentKnowledge(secondStrategy) > medium
AND studentKnowledge(thirdStrategy)  < medium
THEN present exercise-for(thirdStrategy)
```

These rules do not refer to the agents yet.

Choice of Agents. If the goal is to most rapidly prove a conjecture and deep learning is unimportant (as in the pure proof assistant situation), then for every method an agent could be selected for the configuration that checks for applicability and fires, when the method is applicable. However, this is not a typical goal in learning to prove. Rather, learning would involve to understand why a method is applicable, what a particular method is actually doing, and for which purpose it is applied.

For instance, some exercises for group homomorphisms are designed to learn properties of a group, e.g., associativity of the group operation, existence of a unit element and existence of inverse elements. All of them are suggested and the user has to choose the appropriate one. The method agents check whether the current goal contains elements of a group, instead of checking the applicability of each method.

The set of agents is chosen depending on the student model including the learner's history and the overall material. For instance, no learning material for

a novice student would start with an example or exercise in which misleading suggestions for applying a method M are made (also-faulty-configuration). Such exercises would be placed after some positive experience in order to help the student to discover misconceptions and use it productively overcome them. This can be expressed by a heuristic configuration rule like

```
IF  successful(M) in 2 exercises with only-correct-configuration
AND notyet M with also-faulty-configuration
THEN present counter-exercise-for(M) with also-faulty configuration
```

As a result, an exercise in which M cannot be applied is suggested together with an agent that suggests M anyway. These rules are not yet implemented but show the direction.

The configuration module has to be separated from the actual proof planner. During the generation of user-adapted learning material (including interactive exercises) ActiveMath queries the student model and evaluates pedagogical rules that describe which material should be presented and how depending on values in the student model. For instance, it generates material with exercises using a method M whose difficulty is appropriate for the student's mastery level of the concepts involved in the exercises.

3.4 Future Work

For interactive proof planning the methods must be introduced and the student has to learn them, e.g., from an ActiveMath course. This is still unusual in teaching mathematics, where methods appear rather implicit inside of proofs and have to be extracted by the student. However, empirical results suggest that learning proof planning methods can be a learning approach that is superior to the traditional learning for proof [8]. So, we will support the student in learning methods, with a help button in the interaction console that links to a description of the method in ActiveMath's dictionary. We will also extend the feedback and help available in the future.

The current implementation of configurations allows to have agents which make faulty suggestions but these agents do not yet cover all important kinds of possible misconceptions. For example, in most proof planning strategies there is a method for definition expansion which has a parameter for the defined concept. The user could remember a wrong definition or could have forgotten an important part of the definition but the application of the method for definition expansion will always introduce the correct definition. Therefore we want to experiment with extended possibilities for the interaction to detect more deficiencies of a learner. For example, the parameter window for definition expansion could ask the user to provide the definition and compares it to the output of the method that performs the definition expansion.

4 Conclusion

We described the implementation of an agent-based interaction console to be used for interactive proof planning during mathematical proof exercises. It realizes the following features that are useful for learning:

- bearable complexity
- dynamic suggestions for interaction rather than predefined problem solving strategies
- among others, faulty suggestions can be used to trigger learning from failure
- modeling pedagogical strategies by configuration comprising sets of agents and different proof planning strategies
- help for next proof steps provided by automatic proof planning

References

1. V. Aleven and K.R. Koedinger. Limitations of student control: Do student know when they need help? In G. Gauthier, C. Frasson, and K. VanLehn, editors, *International Conference on Intelligent Tutoring Systems, ITS 2000*, pages 292–303. Springer-Verlag, 2000.
2. C. Benzmüller and V . Sorge. ΩANTS– an open approach at combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *Proceedings of the Calculemus Symposium 2000*, St. Andrews, UK, 6–7 August 2000. AK Peters, New York, NY, USA.
3. A. Bundy. The use of explicit plans to guide inductive proofs. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120, Argonne, 1988. Springer.
4. M. Kohlhase. OMDoc: Towards an internet standard for the administration, distribution and teaching of mathematical knowledge. In *Proceedings Artificial Intelligence and Symbolic Computation AISC'2000*, 2000.
5. H. Mandl, H. Gruber, and A. Renkl. *Enzyklopädie der Psychologie*, volume 4, chapter Lernen und Lehren mit dem Computer, pages 436–467. Hogrefe, 1997.
6. A. Meier, M. Pollet, and V. Sorge. Comparing approaches to the exploration of the domain of residue classes. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):287–306, October 2002. S. Linton and R. Sebastiani, eds.
7. E. Melis, J. Buedenbender, E. Andres, A. Frischauf, G. Goguadse, P. Libbrecht, M. Pollet, and C. Ullrich. ACTIVEMATH: A generic and adaptive web-based learning environment. *Artificial Intelligence and Education*, 12(4):385–407, winter 2001.
8. E. Melis, Ch. Glasmacher, C. Ullrich, and P. Gerjets. Automated proof planning for instructional design. In *Annual Conference of the Cognitive Science Society*, pages 633–638, 2001.
9. E. Melis and A. Meier. Proof planning with multiple strategies. In J. Loyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L.M. Pereira, and Y. Sagivand P. Stuckey, editors, *First International Conference on Computational Logic*, volume 1861 of *Lecture Notes on Artificial Intelligence*, pages 644–659. Springer-Verlag, 2000.

10. E. Melis and J.H. Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 115(1):65–105, November 1999.
11. K. Reiss, J. F. Hellmich and F. Thomas. Individuelle und schulische Bedingungsfaktoren für Argumentationen und Beweise im Mathematikunterricht. In M. Prenzel and J. Doll, editors, *Bildungsqualität von Schule: Schulische und außerschulische Bedingungen mathematischer, naturwissenschaftlicher und überfachlicher Kompetenzen. Beiheft der Zeitschrift für Pädagogik*, pages 51–64. Beltz, Weinheim, 2002.
12. J. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C. Wirth, and J. Zimmer. Proof development with omega. In A. Voronkov, editor, *Proceedings of the 19th Conference on Automated Deduction (CADE-19)*, pages 143–148, Copenhagen, Denmark, 2002.
13. K. VanLehn, S. Siler, C. Murray, T. Yamauchi, and W.B. Baggett. Human tutoring: Why do only some events cause learning? *Cognition and Instruction*, 2001.

Formal Proof Authoring: an Experiment

Laurent Théry

Dipartimento di Informatica
Università di L'Aquila, Italy

Laurent.Thery@di.univaq.it

Abstract. In this paper we propose a proof format to write formal proofs motivated by a formalisation of floating-point numbers. This proof format aims at being adequate for both proof presentation and mechanised proof checking.

1 Introduction

The motivation of this work comes from a long-term collaboration with researchers in computer arithmetic. The goal of this collaboration was to apply theorem proving technology to validate some new algorithms for floating-point numbers. The system we have been using is the Coq proof assistant [13]. We first developed a library for floating-point numbers. Then we have been using this library to formally prove some properties of new algorithms that were manipulating expansions, i.e. sorted lists of floating-point numbers. Early results of this collaboration were reported in [7].

In the computer arithmetic community researchers are rather used to write proofs on paper to demonstrate the correctness of their new algorithms. There is a well-established IEEE standard that describes how floating-point numbers should be represented. So people usually take full advantage of the properties of this representation to apply ad-hoc optimizations without jeopardizing portability. Since the standard includes different categories of numbers (normal, subnormal, *NaN*, ...), the correctness proofs are often intricate and require a great deal of tedious checking for a skeptic reader. An example of such a proof is the 19 pages long proof given in [9].

Theorem proving systems are known to be good for checking tedious proofs. Early works [12, 16, 19] have already illustrated the benefit one can get by using theorem provers in the particular setting of floating-point numbers. In that respect, our experiment has only reconfirmed this fact. However, it has also shown some limitations of this approach. Proof systems still need to get further improved in order to make mechanised proofs a natural companion to proofs on paper. From our point of view, an aspect deserves particular attention: the representation of proofs. In systems like Coq proofs are represented by proof scripts. A script is composed of a set of elementary commands called tactics. Tactics guide the prover in the search of the formal proof. Due to their nature, proof scripts are very different from the usual proofs on paper. We believe that this

simple fact makes proof systems more difficult to use for people already familiar with writing proofs. In this paper we address this issue.

The paper is structured as follows. In Section 1 we explain how proofs on paper and proof scripts interact in our experiment. The goal of this discussion is mainly to motivate our new format for formal proofs. This format is given in Section 2. In Section 3, we present a prototype system that aims at providing a user interface to build proofs according to such a format.

2 Proofs on paper versus proof scripts

When starting from scratch, a formalisation in a prover is very often tedious and always time-consuming. All notions have to be defined from basic principles and then all the usual properties have to be derived from these initial definitions. Only after that, it is possible to tackle some applications. This initial formalisation amounts to carefully choosing the initial definitions and then finding an appropriate road to capture all the usual properties. Some experiment is needed in order to know which definitions work better in a specific prover. In that respect, the situation is not so different from programming languages where an experienced programmer knows how to do things.

When starting the collaboration, the people in computer arithmetic we were working with knew very little about theorem proving systems. They were just aware that these systems could be successfully applied to their area as reported in [12, 19]. To our surprise, the definitions to be introduced into Coq, which properties should be proved first, and how they should be expressed were actively discussed. Our fear was that the formal and sometimes very syntactic aspect of theorem proving would impede the collaboration at such an early stage. This was not the case. Moreover, if at the beginning proofs were sketched using the usual definitions, quickly the specific Coq representation was adopted and elegant short cuts to prove some basic properties were even proposed. One explanation of this quick adaptation is that people are already familiar with playing with different presentations of the same problem. For example, different books about the same topic usually present different approaches. In that respect, developing a formalisation in a prover has lot in common with writing a book.

When we passed to actually using the newly developed library to prove the correctness of some algorithms, the nature of the collaboration changed. Before the focus was more on the formal statements of the properties and on the structure of the whole library. Afterwards, when proving the correctness, the main interest was on the proof. It was only at this stage of the collaboration that we realized that mastering how to do proofs in Coq would have a very steep learning curve. In contrast, how to express properties in the logic of Coq was quickly assimilated. Our modus operandi to check proofs in Coq has nearly remained unchanged since the beginning. First, the proof was written carefully on paper with as many details as possible. Then, this document was used as a guide to produce the script for the proof system. The translation was not faithful in the sense that we often took advantage of the deductive power of Coq to restructure

the proofs, mostly to shorten the scripts. In that respect, the proofs on paper were detailed enough so to be sure that in Coq we would only concentrate on writing the scripts and in no case on figuring out how to prove a given subgoal. Then, the proofs on paper and the scripts were taking different roads. The proofs on paper were reworked, mainly shortened so to be appropriate for insertion in some technical report. The scripts were also manipulated after the first script has been obtained. Good practice recommends to reorganize scripts in order to increase reusability and robustness.

When shown how proofs were actually entered in Coq, the first reaction of the people we were collaborating with was that Coq proofs were awkwardly written the other way around. This was true, when translating paper proofs, we were always starting from the bottom of the text. This is of course because of the goal-directed flavour of most tactics. Furthermore, as explained before, the difference between the proof that is published and the proof script that is stored in the formal development is more likely to be even larger. This has at least three unpleasant consequences. First of all, it is not exactly true that the published proof has been formally checked by a theorem prover. What has been really checked is the conclusion of the proof. In particular, nothing ensures that the published proof does not contain errors. Fundamental errors are most likely to have been discovered during the checking in the prover. Still, having just a single stupid error in a published proof is very annoying specially when the main claim of the paper is that all the results have been mechanically checked.

The second consequence is that published proofs are of limited interest for maintaining the formal development. As provers are constantly evolving, keeping the scripts up to date is a real problem. The published proofs tell very little on how the corresponding proof scripts are structured. Being able to quickly find out what a specific subpart of the script is supposed to do is what is really needed when maintaining scripts.

The third consequence that is closely related to the second one is a lack of flexibility. It is often the case that, when something has been proved under particular assumptions, one would like to see the effect of slightly modifying them. Typically in floating-point arithmetic, one could try to change the base of the arithmetic, or the rounding mode. Doing such experiments with a mechanized proof is very simple. One simply needs to change the assumptions and rerun the scripts. The fact that a proof is not valid anymore is detected when the application of a tactic fails. Finding out to which actual step of the paper proof this corresponds is not immediate and requires some non-trivial expertise.

3 Proof format

In order to find an adequate proof format that would reduce the gap between the structure of proofs on paper and the structure of proof scripts, we had three sources of inspiration. The first one is the paper [14] by Leslie Lamport that describes how formal proofs should be written. Reading this paper, it is clear that the main feature of formal proofs is the *structure*. A formal proof

should explicitly expose how the initial problem is decomposed into elementary subproblems and so on until obvious statements are obtained. One of the main requirements of a formal proof format is then to highlight the proof structure.

The second source of inspiration comes from the reading of different proofs in arithmetic. Because they are mostly dealing with transforming inequalities, the proofs can often be understood just by reading the formulae and forgetting the text around. This observation echoes a personal anecdote. When learning mathematics, our teacher in probability was referring to some Russian books for his course with the sentence "You will see, you get used to Russian very quickly". And in fact it was true. With very little knowledge of the language, it was "somewhat" possible to read mathematics books in Russian. In our format, we push this observation to the extreme and take as an assumption that only formulae matter.

The last source of inspiration is the verification condition generator Why [10]. It is used to prove the correctness of programs. For this, the program is annotated with logical assertions. When run on the annotated program, the Why tool generates a list of conditions. Proving all these conditions ensures that the annotations in the program are valid. We would like to have a similar mechanism for our proof format. A proof written in our proof format could be checked by running a tool and proving the resulting conditions. In program verification, the tools generating the conditions use elaborate techniques such as the computation of the weakest preconditions. For proofs, the generation is far more simpler. It amounts to figuring out for each step of the proof which proposition is proved under which assumptions. Only some basic slicing technique is needed.

Adding this layer of generation is important. First of all, it makes it possible to write the whole proof without actually using a proof system. Mechanically checking the proofs is viewed as a separate activity. Note that in the case of program verification one expects that the prover would be able to discard automatically at least 80% of the conditions, we expect the same would hold in our case. Second, the generation creates an explicit link between what is in the proof and what is proved in the prover. Finally, having this phase of generation makes it possible to have a format that is independent of a particular prover. In Why the conditions can be output for different proof systems (Coq, Pvs, and Hol Light) and adding a new output is relatively simple. We would like the same to hold for our format.

The definition of the actual proof format derives naturally from these considerations. An appropriate representation of proofs to highlight their structure is known since decades. It is the *natural deduction style* proposed by Prawitz [18]. For the actual description of the format a natural candidate is XML [3] whose purpose is to exactly represent structured objects. In the following, we split the presentation of the format in two parts. The first part describes how formulae are represented. The second part describes how the proof structure is represented. We then conclude the section by illustrating how the generation works.

3.1 Format for formulae

There already exists a format to represent mathematical objects in XML. It is called MathML [4]. What we present here is a far simpler version that is only interested in representing logical formulae. The encoding is straightforward. A variable is represented by a tag `var`, a subtag `name` and an optional subtag `type`. For example, a variable x of type A is represented by

```
<var>
    <name>x</name>
    <type>A</type>
</var>
```

Each logical operator is represented by a corresponding tag. For example, the formula $\neg(x \vee y) \Rightarrow \neg x \wedge \neg y$ is represented by

```
<imp>
    <neg>
        <or>
            <var><name>x</name></var>
            <var><name>y</name></var>
        </or>
    </neg>
    <and>
        <not>
            <var><name>x</name></var>
        </not>
        <not>
            <var><name>y</name></var>
        </not>
    </and>
</imp>
```

Special tags are associated to functions and predicates. They contain a subtag with their name and the arguments. For example, the predicate $P(f(x))$ is represented by

```
<pred>
    <name>P</name>
    <fun>
        <name>f</name>
        <var><name>x</name></var>
    </fun>
</pred>
```

Finally, each quantifier has its own tag that contains a name and a body. For example, the formula $\forall x. \exists y. P(x, y)$ is represented by

```
<forall>
    <var><name>x</name></var>
    <exists>
        <var><name>y</name></var>
```

```
        <pred>
            <name>P</name>
            <var><name>x</name></var>
            <var><name>y</name></var>
        </pred>
    </exists>
</forall>
```

Three aspects have to be pointed out. First of all, we give the possibility for variables to hold types but types can be omitted. Second, the format for formulae is generic and mostly inspired by the one adopted in [10]. For example, in Coq function application is usually curried and $f(x, y)$ is represented as $((f\ x)\ y)$. Transforming XML formulae into their Coq equivalent is the task of the generator. Finally, logical connectors are n-ary and quantifiers can have an arbitrary number of variables. So for example the formula $x \vee y \vee z$ is represented by

```
<or>
    <var><name>x</name></var>
    <var><name>y</name></var>
    <var><name>z</name></var>
</or>
```

3.2 Format for proofs

As said in our introduction, the source of inspiration is the *Natural Deduction style*. Proofs and subproofs are represented by the tag `proof`. Proofs can be given a name with the tag `name`. The conclusion is denoted by the tag `concl`. For example, the introduction rule for the conjunction

$$\frac{A \qquad B}{A \wedge B}$$

is represented in our format as

```
<proof>
    <name>and Intro</name>
    <proof>
        <concl>
            <var><name>A</name></var>
        </concl>
    </proof>
     <proof>
        <concl>
            <var><name>B</name></var>
         </concl>
    </proof>
    <concl>
        <and>
            <var><name>A</name></var>
            <var><name>B</name></var>
```

```
        </and>
    </concl>
</proof>
```

Note that the relative positions of the subgoals and the conclusion is not fixed. Putting the conclusion first gives a goal-directed flavour to the proof. What is proved is given before explaining why it holds. Putting the conclusion last gives the more usual forward style. Most of the time a proof on paper is carried out using the forward style, but for key steps like the application of induction the conclusions may be given first. This capability to accommodate both styles is given for free by XML.

Another important aspect is the handling of assumptions. This is illustrated with the rule for case analysis:

$$\frac{A \vee B \quad \begin{matrix}[A]\\ \cdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \cdots \\ C\end{matrix}}{C}$$

and its encoding:

```
<proof>
    <name>or Elim</name>
    <proof>
        <concl>
            <or>
                <var><name>A</name></var>
                <var><name>B</name></var>
            </or>
        </concl>
    </proof>
    <proof>
        <hyp>
            <var><name>A</name></var>
            <name>h1</name>
        </hyp>
        <concl>
            <var><name>C</name></var>
        </concl>
    </proof>
    <proof>
        <hyp>
            <var><name>B</name></var>
            <name>h2</name>
        </hyp>
        <concl>
            <var><name>C</name></var>
        </concl>
    </proof>
```

```
    <concl>
        <var><name>C</name></var>
    </concl>
</proof>
```

In our format assumptions are represented by the tag `hyp`. This tag contains the formula that is assumed and optionally the name of the assumption. Note that the rules of natural deduction are only used to illustrate our format. The format is flexible: no limit is put on the number of assumptions and subproofs a proof can hold, so to make it possible to express any proof step.

Another important mechanism is the introduction of local variable names. This is illustrated with the universal introduction rule

$$\frac{P(c)}{\forall x.\, P(x)}$$

It is represented in our format as

```
<proof>
    <name>forall Intro</name>
    <proof>
        <var><name>c</name></var>
        <concl>
            <pred>
                <name>P</name>
                <var><name>c</name></var>
            </pred>
        </concl>
    </proof>
    <concl>
        <forall>
            <var><name>x</name></var>
            <pred>
                <name>P</name>
                <var><name>x</name></var>
            </pred>
        </forall>
    </concl>
</proof>
```

As for assumptions and subproofs, more than one local variable can be introduced in a proof.

The last mechanism we need to complete our format is the justification. It gives the possibility of explicitly referring to theorems, local assumptions or local proofs in the proof. Justifications are represented by the tag `justification`. This tag contains a list of names. For example, the expression

```
<proof>
    <name>prop2</name>
    ...
    <justification>
```

```
        <name>prop1</name>
    </justification>
</proof>
```

defines a theorem *prop2* that makes use of another theorem *prop1*. It allows one to explicitly provide the dependence in the proof construction. We have extended the tag `justification` with the capability of putting some proof script:

```
<proof>
    <name>prop2</name>
    ...
    <justification>
        <prover name="Coq">...</prover>
    </justification>
</proof>
```

The `prover` tag is optional and gives the possibility of having a document that can be checked automatically. Having several `prover` tags inside the same justification makes the document checkable by different provers.

The format as it stands satisfies all the requirements expressed in the previous section. It contains structure information and formulae. To make it human-readable, we allow text to be inserted inside tags. For example, the previous proof based on case analysis can be commented with the following texts:

```
<proof>
    <proof>
        <concl>
            We have
            <or>
                <var><name>A</name></var>
                <var><name>B</name></var>
            </or>
        </concl>
    </proof>
    By case analysis
    <proof>
        Case 1:
        <hyp>
            Assuming
            <var><name>A</name></var>
            (<name>h1</name>)
        </hyp>
        <concl>
            We get
            <var><name>C</name></var>
        </concl>
    </proof>
    <proof>
        Case 2:
        <hyp>
            Assuming
```

```
            <var><name>B</name></var>
            (<name>h2</name>)
        </hyp>
        <concl>
            We get
            <var><name>C</name></var>
        </concl>
    </proof>
    <concl>
        In both cases, we have
        <var><name>C</name></var>
    </concl>
</proof>
```

Using a simple style-sheet presentation we get:

```
We have A\/B
By case analysis
Case 1:
  Assuming A (h1)
  We get C
Case 2:
  Assuming B (h2)
  We get C
In both cases, we have C
```

This example is of course too simplistic and just illustrates how the format could be made easily readable. The presentation content of our format needs to be further developed to meet the standard of scientific publication.

3.3 Generation

The algorithm to generate conditions is straightforward. It has only been implemented for Coq but we believe that it could be easily adapted to other systems. It consists in a traversal of the XML structure from top to bottom and from left to right. Each time a `proof` tag is encountered, its subproofs are first recursively processed, then a condition is generated for it. Each condition is represented by a lemma and a piece of script is added to do the necessary book-keeping and get the appropriate assumptions. To give a more concrete example, consider the following proof script

```
<proof>
    <hyp>
        <var><name>A</name></var>
        <name>h1</name>
    </hyp>
    <proof>
        <name>p1</name>
        <concl>
            <var><name>B</name></var>
```

```
            </concl>
        </proof>
        <proof>
            <name>p2</name>
            <hyp>
                <var><name>C</name></var>
                <name>h2</name>
            </hyp>
            <concl>
                <var><name>D</name></var>
            </concl>
        </proof>
        ....
    </proof>
```

The first condition that is generated is relative to `p1` and is represented by a lemma `c_p1`. It has the following form:

```
Lemma c_p1: A -> B.
  Intros h1.
  Apply ok.
Qed.
```

The subproof `p1` has been declared in a context where the assumption A is visible. Its conclusion is B. So the statement of the conclusion is $A \Rightarrow B$. The first task of the script is to introduce assumptions. In Coq, assumptions can be named. So we can faithfully reflect the name given in the document. In our example, the tactic `Intros h1` introduces the first assumption A with the name `h1`. The final tactic `Apply ok` is generated so that the proof is always accepted by the prover. This tactic represents the application of a predefined axiom `ok` that simply states that everything is true. This simple trick gives us typechecking for free. Any error in the initial document is automatically detected by the prover when processing the generated file. As each lemma represents a single step in the proof, tracking the origin of the error in the initial document is easy. Finally the task of actually checking the proof is performed by replacing all the applications of the axiom `ok` with appropriate tactics in the generated file.

For the second subproof `p2`, the condition has the following form:

```
Lemma c_p2: A -> C -> D.
  Intros h1.
  Generalize (c_p1 h1); Intros p1.
  Intros h2.
  Apply ok.
Qed.
```

There are two assumptions in the statement of the lemma: A comes from the context and C is a local assumption of `p2`. For the proof script, the tactic `Intros h1` introduces the first assumption A of the lemma with the name `h1`. The second tactic is more elaborate. In order to introduce `p1`, it needs to instantiate `c_p1` with the context that is common between `p1` and `p2`. Here the common context

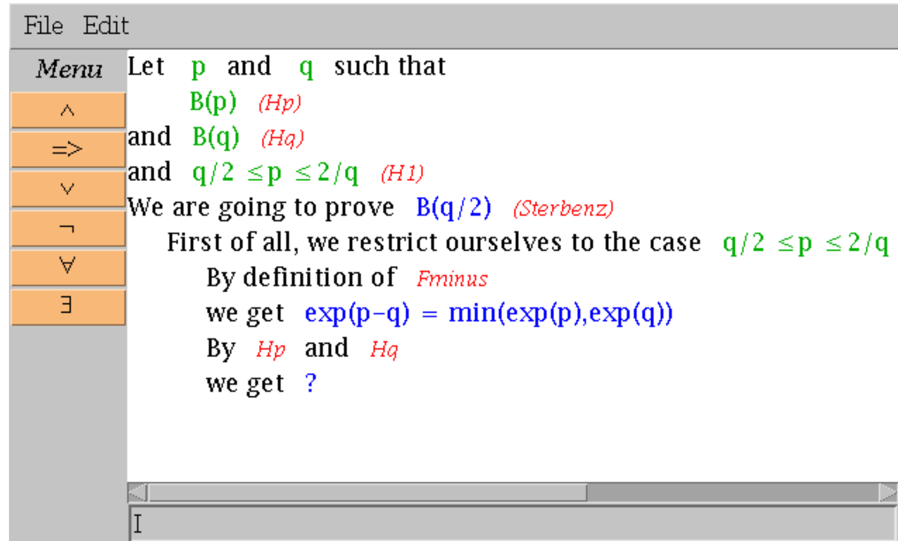


Fig. 1. A snapshot of the prototype interface

is the assumption A whose name is `h1`. The expression `(c_p1 h1)` corresponds to a proof of B. The tactic `Generalize (c_p1 h1)` generalizes the goal with B. Before this tactic, the goal is $C \Rightarrow D$, after it is $B \Rightarrow C \Rightarrow D$. The tactic `Intros p1` introduces B with the name `p1` in the assumption. The result of the two tactics is the expected one: the fact that B is true can be used in the proof of `p2` with the assumption `p1`. The third tactic simply introduces C with the name `h2` in the assumption list. Note that we take a special care in introducing assumptions in the proper order. The assumptions introduced last are the ones displayed next to the conclusion. In the generation we ensure that this proximity is faithful to what is present in the initial proof document. In our example, the closest assumption is `h2`, then `p1` and then `h1`.

Finally, when all introductions are done, we possibly reorganize the order of the assumptions by taking into account the references given in the justifications. This improves the performance of automatic tactics that usually privilege the last introduced assumptions.

4 Prototype interface

To experiment with our proof format, we have been developing a prototype interface. An early evaluation of the editing tools for XML such as Xerlin [24] shows that these generic tools are often difficult to customize. As we wanted to be able to quickly test different ideas, we built our application on top of the AÏOLI toolkit [1]. A snapshot of the current state of our prototype is given in Figure 1. The application is composed of a window with 4 components:

- At the top of the application, a menu bar proposes the basic operations. The File pulldown contains buttons for reading and writing files and for generating the output. The Edit pulldown contains buttons for the usual editing operations: copy, paste, cut, insert.
- In the center, a text region displays the current state of the document.
- On the left, a dynamic list of buttons is proposed to guide editing. Which button is proposed depends on which part of the document is selected.
- At the bottom, a text buffer lets the user enter the text to be inserted in the document.

For displaying the current state of the document, the direct XML format is clearly inadequate, as it is too verbose. To get a more concise presentation, we follow some basic principles:

- Indentation is used to show the proof structure. The presentation of subproofs is shifted of two characters to the right with respect to the presentation of the proof.
- The tags `var`, `hyp` and `concl` are represented using different colours: two different kinds of green for the first two, blue for the last one. It makes the distinction between what is assumed (green) and what is proved (blue) immediately accessible.
- Formulae are displayed with their usual pretty-printing form.
- Names of proofs, assumptions, and references have a special font (italic) and a special colour (red).

This default configuration can be modified but the basic principles (indentation, colour differences) need to be preserved.

Even though the document is represented as a text, the application keeps track of the relation between what is displayed on the screen and the XML document. This connection is mainly used to guide the mouse selection. The selection is structured: only tags can be selected. For example, when dragging the mouse to perform a selection, the region that is selected corresponds to the smaller tag that contains the initial point of the drag action and the current point.

The actual process of editing is performed in a structured way. Initially the document is composed of a single place-holder. This single place-holder is then progressively refined to get the whole document. To guide the editing process, a menu proposes different items. Available items are declared in a configuration file. During the editing process, only the items that are relevant to the selected region are proposed to the user. An item declaration is given by a name, a pattern and an action as follows:

```
name: pattern -> action
```

To express pattern and action, the usual XML syntax is used. A simple extension is needed to denote place-holders, arbitrary tags in pattern, and selection. For a place-holder, the notation is `<|`*name*`|>` where *name* is the name of the place-holder. Most of the time `<|?|>` is used. In patterns, an arbitrary tag is denoted

by `<|*`*name*`|>` where the name *name* is used to refer to this tag in the action description. It is also possible to refer to an arbitrary list of tags using the notation `<|**`*name*`|>`. An action is simply the XML expression that should replace the expression that has been filtered. In this action, place-holders and the variable of the pattern can be used. For example, the item

```
and:
  <|*x|> -> <and> <|*x|> <|?|> </and>
```

has a pattern that filters any tag and build a conjunction with the filtered term and a place-holder. Note that the action also imposes some restrictions on the applicability of the pattern. So this item is only proposed when the selected tag corresponds to a formula.

While editing it is also convenient to move automatically the selection. This is expressed syntactically in the declaration by putting double brackets around the tag that has to be selected in the action. For example, refining the previous item

```
and:
  <|*x|> -> <and> <|*x|> <<|?|>> </and>
```

indicates that the selection should be on the new place-holder. Selection can also be used in the pattern to give contextual filtering. For example, the item

```
and:
  <and> <|**x|> <<|*y|>> <**z> </and>
  ->
  <and> <|**x|> <|*y|> <<|?|>> <|**z|> </and>
```

makes it possible to capture the special case where the tag was already in a conjunction. Note that these two item declarations have purposely the same name. In the menu, items are represented by their names and a given name occurs at most once. When the user selects a name, it is the action with the most specific filters that is triggered. This generic mechanism based on rewriting rules provides a powerful yet simple way to customize the editing menu. For formulae, an adequate menu coupled with the capability of easily copying and pasting formulae, thanks to the structured selection, has proved to be a very effective way for building formulae quickly.

For building proofs, item declarations are mostly used to create short cuts for the most useful proof steps: proof by induction, proof by cases, proof by contradiction. More elaborate guided editing could be developed. Ultimately we could think of integrating powerful external components, such as theorem provers or computer algebra systems, to guide editing. Connecting a theorem prover would make it possible to get the usual interactive proof editing mode, where the prover fills in the formulae of the subgoals. Connecting a computer algebra system would give access to computation and simplification of formulae.

5 Related Works

We are aware that most of our problems came from the simple fact that the tactic language in Coq has been thought as a little programming language whose goal is to build proofs. As in programming, conciseness and genericity in proof scripts are then privileged. This is not particular to Coq but common to all provers based on tactics. Scripts in such provers usually contain very few formulae. As coined in [15], adding formulae in script increases the *viscosity* of the script. It reduces the reusability of scripts and make them less robust to modifications. This simple fact that formulae do not usually appear in proof scripts shows how inadequate scripts are to reflect the usual proofs.

There have been attempts to get a more natural language to interact with provers. The first and most impressive one by far is the Mizar project [17]. Other interesting attempts include [20, 22, 25]. Following the terminology used in [11], these systems propose a declarative style of proving, while systems like Coq offer a procedural approach. Declarative scripts usually contain lots of formulae and are then closer to proofs on paper. Unfortunately these systems impose some strong restrictions on the way proofs should be written. In Mizar for example, the proof has to be given with the level of detail imposed by the system. As the system has very little automation, proofs script are often too detailed for a human reader. In Isar, some basic constructs are hard-wired. An example is the proof by cases, where the presentation of the different cases in the document has to follow the exact order in which the object was declared.

Other interesting approaches include attempts to accommodate both procedural and declarative styles [8, 23], extract proof texts from tactics [5], extract proof texts from proof objects [2, 6]. Note that when provers have proof objects, it would be possible to convert automatically proof objects into our format in a very similar way as in [2, 6]. The result would most probably be far too detailed. However, with some support for improving the presentation while keeping the proof script consistent, this reverse engineering activity could be an effective way to get readable proofs.

6 Conclusion

In this paper we have presented a very simple and flexible format for writing formal proofs. This format is independent of any theorem prover. Writing proofs is meant to be as natural as possible. With respect to the usual way of writing proofs, the writer is only asked to explicitly indicate the proof structure. At each step it is then clear what the assumptions are and what the conclusion is. We have also presented some ideas on how a user interface could help writing proofs in this format.

The separation of the activity of proof writing from the activity of proof checking has been motivated by our own experiment. We do not claim that this separation should always be the rule. In our case, it makes it possible to conciliate the necessity of publishing the proof in a human-readable form with the tedious

task of mechanically checking the proof in all the very details. In that respect we consider the mechanic checking of the proof just as a way to increase the confidence in the proof and not in any case as a way to substitute the refereeing process.

In Section 2, we have described three drawbacks of the usual loose connection between the published proof and its machine-checked version. The first drawback was that the computer proof usually only checks the final statement of the published proof. In our framework we get a tighter connection. Every step of the published proof has been checked: for each step there is a corresponding lemma in the formal development. The second drawback was the difficulty of maintaining proofs. To ease maintenance a good practice is to always split big proofs into smaller pieces. Our generating process enforces this practice: every lemma only covers a single step of the proof on paper. The last drawback was the difficulty to experiment with slightly different versions of the final statement. In our case, the variations can be done directly on the published proof by changing the statement. When rerunning the proof script, the lemmas that the prover fails to reestablish directly indicate the steps that need to be fixed in the published proof.

Some more work is still needed in order to turn our experiment into a realistic approach. For example, the conditions that are generated are rarely provable automatically by Coq. More tactics need to be developed in order to get a reasonable ratio of conditions proved automatically. Once this has been done, we will still need to convince people to write proofs in our format. The fact that we are using XML could be a problem. Currently very few scientific publications are written directly using this technology. The situation may change in the future. An interesting alternative is to adapt our ideas to the TeXmacs [21] system. This system proposes a very effective environment to write LaTex documents. It has a large community of users that could be interested in this light connection with theorem provers.

References

1. Aïoli. A Toolkit to Build Interactive and Symbolic Applications, Available at `http://www-sop.inria.fr/lemme/aioli/`.
2. Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. HELM and the Semantic Math-Web. In *TPHOLs'01*, number 2152 in LNCS, pages 59–74, Edinburgh, Scotland, 2001.
3. Tim Bray, Jean Paoli, and C. Michael Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation.
4. David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier. Mathematical Markup Language (MathML) version 2.0. W3C Recommendation.
5. Avra Cohn. Proof Accounts in HOL. unpublished draft. 1988.
6. Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. In *TLCA*, volume 902 of *LNCS*, pages 109–123, Edinburgh, Scotland, 1995.
7. Marc Daumas, Laurence Rideau, and Laurent Théry. A Generic Library for Floating-Point Numbers and its Application to Exact Computing. In *TPHOLs'01*, number 2152 in LNCS, Edinburgh, Scotland, 2001.

8. David Delahaye. Free-Style Theorem Proving. In *TPHOLs'02*, number 2410 in LNCS, pages 164–181, Hampton, VA, USA, 2002.
9. James Demmel and Yozo Hida. Accurate floating point summation. Available at `http://www.cs.berkeley.edu/~demmel/AccurateSummation.ps`, 2003.
10. Jean-Christophe Filliâtre. Proof of Imperative Programs in Type Theory. In *TYPES '98*, volume 1657 of *LNCS*, Eindhoven, Netherlands, 1998.
11. John Harrison. Proof style. In *TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996.
12. John Harrison. A Machine-Checked Theory of Floating Point Arithmetic. In *TPHOLs'99*, number 1690 in LNCS, pages 113–130, Nice, France, 1999.
13. Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq Proof Assistant: A Tutorial: Version 6.1. Technical Report 204, INRIA, 1997.
14. Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1995.
15. Nicholas Merriam and Michael Harrison. What is wrong with GUIs for theorem provers? In *UITP '97*, INRIA Report, Sophia-Antipolis, France, 1997.
16. Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA, Langley Research Center, 1995.
17. Mizar. *Journal of Formalized Mathematics*. `http://mizar.org/JFM/`.
18. Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.
19. David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.
20. Don Syme. Three Tactic Theorem Proving. In *TPHOLs'99*, number 1690 in LNCS, pages 203–220, Nice, France, 1999.
21. TeXmacs. A Free Scientific Text Editor, Available at `http://www.texmacs.org/`.
22. Markus Wenzel. A Generic Interpretative Approach to Readable Formal Proof Documents. In *TPHOLs'99*, number 1690 in LNCS, pages 167–184, Nice, France, 1999.
23. Freek Wiedijk. Mizar Light for HOL Light. In *TPHOLs'01*, number 2152 in LNCS, pages 378–393, Edinburgh, Scotland, 2001.
24. Xerlin. Opensource Extensible XML Modeling Application, Available at `http://www.xerlin.org/`.
25. Vincent Zammit. On the Implementation of an Extensible Declarative Proof Language. In *TPHOLs'99*, number 1690 in LNCS, pages 185–202, Nice, France, 1999.

Thoughts on Requirements and Design of User Interfaces for Proof Assistants

Norbert Völker

University of Essex, England
norbert@essex.ac.uk

Abstract. This position paper discusses various issues concerning the requirements and design of proof assistant user interfaces (UIs). It starts off with an analysis of some of the difficulties faced by UI projects in academia. Reusability is stressed as the key to provide high quality user interfaces for more proof assistants. In order to avoid being influenced by particular provers, a generic description of proof assistant interaction is given. This is followed by an exposition of use cases and object identification. The usefulness of these requirement elicitation techniques in the theorem proving domain is demonstrated by identifying some use cases which are lacking support in a current UI/proof assistant combination.

The second half of the paper begins with a discussion of human computer interface (HCI) design issues in the theorem proving context. After a brief review of the "GUI versus text mode" debate, suggestions are made how the usability of future UIs could be improved by application of the "principle of least effort", better use of GUI facilities and better customisation capabilities. The paper ends with a consideration of architecture and system design issues. In particular, it argues for a platform architecture with an extensible set of components and the use of XML protocols for communication between UIs and proof assistant backends.

1 Introduction

Theorem proving environments such as COQ [1], HOL [2], Isabelle [3], Nuprl [4], Omega [5], PVS [6] and others [7] are becoming increasingly popular for the formalisation of derivations in different areas of computer science and mathematics. Some of these systems are associated with user interfaces (UIs) that try to alleviate the strain of dealing with a complex proof assistant. Recent noteworthy achievements include ProofGeneral [8], PCOQ [9], LOUI [10] the Nuprl interface and IsaWin [11]. Nonetheless it is fair to say that progress in user interfaces has been slow compared to advancements in the proof assistants themselves.

The aim of this paper is to aid the development of future UIs by discussing important issues concerning requirements elicitation, human computer interface (HCI) design, architecture and system design. A lot of emphasis will be put on reusability and adaptability as these are seen as key design goals.

2 Proof Assistant Interfaces: Problems and Reusability

Most leading proof assistants have been around in some form or other for a decade or longer. Several have matured to a point where they are suitable for mechanical proof checking of substantial derivations.

The gradual development of proof assistants is in contrast to the more haphazard history of theorem prover user interfaces. A look at the literature from the 1990s reveals several projects which seem to have had little impact. In some cases, the projects produced systems that never advanced beyond rather small groups of users. It is interesting to see that among the more popular survivors are several (X)Emacs based systems, i.e. ProofGeneral and the user interfaces for PVS and IMPS [12].

It order to increase the impact of future proof assistant UI efforts, it is useful to be aware of some of the difficulties facing such projects.

- Designing a good user interface for any complex system is a demanding task that requires good skills both in HCI and program design.
- Implementing a user interface can be a tedious job that is less intellectually stimulating than developing the proof assistant itself. It is easy to underestimate the necessary efforts because of the apparent visual simplicity of the product.
- Systems that are build on top of a platform such as (X)Emacs require relatively less coding. However, they can be plagued by platform restrictions and problems which are out of the developer's control. Another drawback is the fact that the addition of features to a platform sometimes requires coding in an uncommon scripting language. This increases the threshold for contributions from developers outside a core team.
- There is little reward for building and maintaining user interfaces. Because of the lack of commercial interest, most developers are in fact academics or students. Members of both of these groups are not rewarded directly for perfecting and maintaining user interfaces but only rather indirectly, namely in so far as this contributes to their research success, teaching, and in case of students the award of qualifications.
- Most theorem provers have been developed in functional programming languages from the ML and Lisp families. Compared to other languages such as C++, Java or Basic, these functional languages are lacking in graphical user interface (GUI) toolkits and in some cases provide only poor support for light-weight concurrent processes. These problems have lead to several projects that connect functional languages with frameworks that have better GUI facilities, see for example sml_tk [13] and recent O'Caml [14] bindings for TK and GTK+. Still, the resulting development environments are more cumbersome than the direct use of a modern commercial GUI building tool.
- Theorem provers are still rapidly evolving and so are user interface technologies. This increases the required maintenance effort.

For the reasons mentioned above, only a few academic institutions have the necessary resources to build and maintain completely custom UIs. Even these

institutions would benefit from more reuse by freeing up developer time. This suggests that it is in the interest of the academic theorem proving community to devise reusable UI components and UIs.

3 Proof Assistant Interaction

In the past, with the notable exception of ProofGeneral, most UIs have been targeted at a particular proof assistant. Unless great care is taken, this leads to a design with a tighter than necessary coupling between the UI and the proof assistant. This makes reuse difficult. Although doubtlessly influenced by the author's experience with Higher-Order Logic (HOL) theorem proving environments, this paper tries to approach UI requirements from a more general perspective.

The primary role of the UI is to provide support for the development of theories with a proof assistant. Here is an outline of the interaction from a user point of view:

- Development usually takes place within some *project* such as building a library, finding a proof for a mathematical theorem or the formal verification of a program.
- Development is sectioned in a number of *theories* (or "sections"). Theories form a hierarchy with each child theory extending one or more parent theories.
- Theory developments can be made persistent, for example in form of proof script files. The system provides commands to store and load theories.
- All theory development activities take place in a *logical context* consisting of
  1. a *logical basis* that comprises declared constants, axioms, and for typed logics the declared types and possibly sorts.
  2. the set of *theorems* that have already been proven for this logical basis.
- Apart from the logical context, there is also an *extra-logical context* that influences the user's interaction with a system. This includes the setup of parsers and pretty printers, proof tools, documentation generators and the state of display options and the setup of proof tools and documentation generators.
- An interactive *proof* is started by posing a logical formula (the *main goal*) that is to be proven. This is followed by a number of *proof steps*. In each step, the user issues some *proof command*. This is carried out by the prover and leads to a new *proof state*.
- A proof state contains a list of *open goals* that are left to prove. It might also contain other information such as *local declarations and assumptions* within a proof.
- A proof ends when a state is reached that has no open goals.
- Proof attempts can be aborted.
- On some systems, proof activities can be nested, i.e. it is possible to temporarily abandon a *proof attempt* and return to it later after proving other theorems in the meantime.

– Proof commands can refer to proof tools, theorems, subgoals and their subterm, types, terms and other entities.

The term "theory development" does not just denote the expansion of theories with new constants, theorems and other elements. It also refers to activities that concern the modification, inspection and deletion of theory elements, i.e. the inspection of the current proof state, the editing of an existing proof or the deletion of a theorem.

4 The Case for Requirements Elicitation

In the author's experience, academic projects often do not carry out a detailed requirements elicitation. Writing requirements documentation might be perceived as a job which ultimately takes more time than it is worth. This section argues for the usefulness of use cases and object identification as two relatively lightweight means of requirements elicitation.

A Use Case Example

Use case analysis [15] is one of the more successful software engineering technique for collecting requirements. Each use case abstracts over a set of user-system interactions ("scenarios") that are performed by users in pursuit of some aim.

Below is a sample use case for a hypothetical proof assistant system. It is based on a use-case template from [16].

Title of Use Case: Adding a constant to a theory
Actors: Specifier
Preconditions: System is either in top-level or theory mode.
Postconditions (Successful Outcome): Constant has been added to the theory
Main Success Scenario:

1. The specifier requests "Add constant" operation.
2. The system responds with a list of theories for the current project.
3. The specifier selects a theory.
4. The system responds with a display of the constants already declared in the theory and a form for entering the details of the new constant.
5. The specifier enters the constant details and submits the form.
6. The system adds the constant to the theory.
7. The success of the operation is indicated by a status message in the UI.

Alternative Flows

1-3 a. If the system is in theory mode, then the specifier can also request "Add constant to current theory". In this case steps 2 and 3 are omitted.
2b/4 a. The specifier can cancel the process by choosing a "cancel" option.

4 b. If the specifier indicates the theory by string input, then the system checks that the current project contains such a theory. If this check fails, then the system responds with a suitable error message and redisplays the list of theories.
4 c. The system checks that the specifier is authorised to change the logical basis of the theory. If this check fails, then the system responds with a suitable error message and redisplays the list of theories.
6 a. The system checks that the form details specify a valid new constant for the theory. If this check fails, then the system responds with a suitable error message and redisplays the constant details form.
7 a Cancellations and errors are also indicated by status messages.

Special Requirements

- All text input forms should allow copy-and-paste.

Open Issues

- Should there be a syntax-directed editor for input of the constant definition?

Actors

In a use case, the term"actor" refers to a particular role in which users interact with a system. The actor "specifier" above stems from an article by J. Goguen [17] who distinguishes for theorem proving applications between the three actors "specifier", "prover" and "reader". These three can all be seen as sub-roles of a more general "theory developer" role. Other roles in which users might interact with a proof assistant are a "proof tool developer" actor who performs activities such as programming, adaptation and testing of proof procedures as well as the standard "system administrator" actor who performs activities such as installation, monitoring and updating of the system.

Identification of Domain Objects

An important step in requirements engineering is the identification of (classes of) domain objects that are relevant from the user's point of view. A simple starting point to finding such objects are the nouns in use cases and other descriptions of user-system interactions. In our case, object classes can be found by going through the description in Section 3 and by looking at proof assistant manuals. Below is an initial, incomplete list of domain objects that are relevant in the interaction of users with a proof assistant:

- project, theory, theory hierarchy, child theory, parent theory
- logical context, logical basis, extra-logical context;
- axiom, theorem;
- formula, term, constant, type, sort;
- proof, goal, main goal, proof step, proof command, proof state, open goal, proof attempt;

- tools such as parsers, pretty printers, proof editors, natural language processors, etc

For user interface design, the identification of domain objects is helpful because it raises the question how to represent the state of objects from a class in general, independent of a concrete use case.

Finding Use Cases

Domain objects help to structure the search for use cases. For each class of objects, one considers in how far there is a need to support operations from the following generic categories:

- creation of objects,
- modification of objects
- inspection of the state of objects
- making objects persistent
- deletion of objects

Application of this technique quickly leads to a long list of generic use cases for theorem proving environments. As an example, here are some of the use cases that can be identified in this way for the domain class “theorem”:

- Creation of a theorem via an interactive “backwards” proof. Some of the steps of this primary use case were already sketched in Section 3.
- Creation of a theorem directly – without stating a goal – by applying operations to existing theorems.
- Renaming a theorem
- Editing the proof of a theorem
- Modification of the statement that is proven in a theorem. This also requires an adaption of the proof.
- Inspection of a theorem and its associated proof.
- Saving a theorem with its associated proof.
- Deletion of a theorem from a theory.

Here are three more use cases which were identified using these generic considerations and which lack support in the current ProofGeneral/Isabelle combination:

- While Isabelle provides a number of operations to inspect and modify its standard proof tools, there is no support for such actions in ProofGeneral.
- From a user point of view, renaming a constant is a use case that occurs sometimes during the initial phases of the development of a new theory. Neither ProofGeneral nor Isabelle supports this directly. Instead, the user has to edit all occurrences and rerun proof script files manually.
- There is no support for use cases involving projects in ProofGeneral such as creating a new project, moving theories from one project to another one, or “making” a project (running all proofs and associated document generation). In fact, there is no explicit notion of projects in Isabelle - they are identified with directories.

These three use cases illustrate different problem situations. In the first case, the UI support is simply lagging behind the facilities offered by the proof assistant. This could be mended easily by adding custom UI elements that invoke these operations. In the second example, a particular use case is not supported directly by the proof assistant. Hence support in the UI would also require an extension of the proof assistant resp. its infrastructure. In the third case, a whole software engineering concept ("project") is missing from the UI/ proof assistant combination.

The Role of Use Case Analysis

Use cases are a relatively simple and lightweight requirements elicitation technique. Their main purpose is to aid in the clarification of functional requirements from a user point of view. Typically, they are written during the early phases of software projects, be it the development of new systems or the extension of systems with new functionality. Use cases also provide a good starting point for user interface design, testing and – in case of theorem provers much needed – user documentation.

A use case analysis of proof assistants and their UIs should produce the following deliverables:

- A list of domain classes that play a role in user interaction.
- A catalogue of all use cases structured according to the domain classes. This list should be annotated with priorities, for example by distinguishing primary, secondary and optional use cases.
- A detailled description of key use cases based on a template such as used in the "Add a Constant" example above. These descriptions can be structured using "generalisation", "includes" and "extends" constructs [18].
- If desired, a graphical overview of use cases in form of UML use case diagrams can be added.

Use case analysis is of course no panacea that magically solves all problems of requirements engineering. For example, the identified domain classes are not necessarily appropriate as design and implementation classes. This point was made by B. Meyer [19] who also criticised use case for their specification of sequentiality. Another problem can be that use cases might invite a too-low level, concrete view of the system that is too much influenced by similar, already existing system.

Despite these drawbacks, experience with use cases over the last decade has shown that they are a valuable technique for collecting requirements. For the development of theorem proving environments, a special benefit arises from the fact that they take a very user-centric view. This should act as a counterbalance in what otherwise tends to be a prover-functionality driven development. Another strength of use cases is that they identify typical sequences of interactions which need to be supported efficiently as a unit. In the author's experience of proof assistants, supporting whole use cases rather than isolated interactions is a point that does not always seem to be taken into account.

5 Human Computer Interface (HCI) Issues

HCI Design: Principle of Least Effort

The interaction of a user with a proof assistant is an instance of human-computer interaction (HCI) [20]. As such, the usual guidelines of good HCI design apply. One of the most important of them is the "principle of least effort". Here are some steps that can be taken in order to minimise the user effort:

- Auto completion helps to minimise the amount of typing. This is important as theorem prover interaction can require a lot of typing and mouse/pointer operations. This increases the risk of developing repetitive strain injuries (RSI). Auto completion could be used whenever it is necessary to name objects such as theories, theorems, proof commands, etc. Completion should be intelligent in the sense that it takes object types and context into account. An advanced implementation could perhaps even consider the likelihood of different possible completions.
- Menus should offer only possible choices.
- Menu items could be grouped into a recently used/frequently used section and a rarely-used section. The latter section would be visible on demand only.
- Use of suitable default values whenever a choice is offered.
- Allow reuse of previous inputs, i.e. base a new constant definition on editing a similar previous definition or allow editing of a previous proof command.

K. Eastaughffe [21] has identified further HCI design principles for theorem prover support. In particular she suggests complimentary views of proof constructions, ease of undo operations, flexibility in the way users can articulate commands to a prover, and support for concurrent proof constructions.

Graphical Interaction versus Text Mode

Graphical User Interfaces (GUIs) have transformed the use of computers and contribute significantly to their popularity. They are kind to novices and non-experts as they make it unnecessary to learn command languages. GUIs are based on the direct manipulation metaphor: objects are represented by graphical elements such that a manipulation of the graphical representation induces a corresponding operation on the represented object. These days, even most text editors are GUI applications - they usually support graphical operations such as positioning with a pointer device, selecting of items in pull down menus, etc.

Despite all the arguments for GUIs, the case for graphics based interaction rather than text is not always clear cut. Text based interfaces can be more efficient - selecting an item from a large scrolling list takes longer than simply typing its name, especially when auto-completion is provided. In the case of theorem prover UIs, there are conflicting reports. Some users like proof trees [22] such as provided by PVS or Jape while others have commented that – while useful for pedagogical purposes – such trees can be difficult to use for all but the

smallest proofs [17]. One study has even cast general doubt on the usefulness of GUIs for proof assistants [23]. In particular, it mentions the danger of brittle proofs arising from the identification of subterms by pointing and comments that "There is even strong evidence to suggest that ease of interaction tends to reduce planning and lead to poorer user performance in problem solving domains...".

In our view, this controversy suggests that the UI should offer a choice between text based and graphical interaction whenever there is no clear-cut advantage to use either. This applies for example to selections which are often faster by keyboard, at least for expert users. Providing such a choice also caters for individual preferences.

Better Use of GUI Facilities

Because of the large potential benefits, there is a strong case for continuing research into better use of GUI facilities during proof assistant interaction. Especially promising it the use of hyper-linked text (HTML/ XML) as it offers an efficient and relatively easily implementable way of browsing information. Here are some potential applications:

- Theorem search results: link to proofs
- Links from one theorem to other theorems that are closely related
- Browsing proofs at various levels of detail
- Constant in a goal: link to its definition
- Unknown in a goal: link to a box which allows its instantiation

Drag-and-drop operations are useful for manipulating "container" objects and aggregations. In the context of proof assistants, possible applications include:

- Moving a theorem from one theory to another theory
- Assembling theorem sets that parameterise a proof procedure

Proof-by-pointing [24] is an advanced method for graphically constructing proofs. It includes the generation of proof scripts from user interactions, called "proof script management" in [24]. It would be good to see this feature supported in more proof assistant UIs.

Using the same underlying idea, one could also use "clicking" to perform forward or backward reasoning from goal premises or a goal conclusion via a pre-selected theorem. Even simpler would be support for a "proof by clicking" of a subgoal by assumption.

Customisation

In order to increase usability, it is vitally important that proof assistant users can adapt the UI easily in order to accommodate for their own preferences as well as to adapt to changes in the proof assistants. This includes:

- The adaptation of menus, menu items, tool bars, key board short cuts etc. A simple example would be the addition of a tool bar button which generates a particularly frequently used proof command.

– The adaptation of formatting, both of proof assistant outputs as well as UI elements: fonts, font sizes, resizing of windows, pretty-printer settings, colouring, etc.

Note that such adaptations should be context sensitive in the sense that they depend on different UI modes such as top level mode, theory mode or proof mode.

Adapting the system needs to be recognised as a normal activity that deserves proper documentation. While some systems such as XEmacs offer customisation dialogues, it can be easier to directly edit configuration files. The latter is acceptable as long as the format of these configuration data is easily understandable. Unless there are good reasons, XML appears to be the obvious choice as configuration file format for new projects.

Customisation should be possible at different levels such as project level, theory level, proof level with settings on a more specific level overriding those on more general levels, i.e. setting the visibility of types to true in the context of a theory should override a setting of the visibility of types to false on the project level.

6 Architectural and System Design Issues

Paradigms

When developing software systems, it helps to have paradigms and metaphors that guide the design. From the user point of view, the overall system paradigm that we propose for proof assistant UIs is that of a *control center* similar to modern UML/ programming IDEs (integrated development environments). From the system designer point of view, our suggestion is the paradigm of a platform which allows the plugging in of components.

Basic Architecture

The user interface is the "front end" layer of the system architecture of a theorem proving environment. Its role is to permit the user an efficient control of the different functionalities offered by the system.

In many software systems, the user interface is written in the same programming language as the rest of the system. In this case, the lower layers provide interfaces consisting of functions that can be called from the UI in order to carry out user requests.

With most current proof assistants, the situation is different. Rather than providing an Application Programmers Interface (API) in some programming language, the proof assistants offer a text-based interface. This can be used directly by a user and yields a basic console ("command line") HCI. In order to obtain a more advanced user interface, one builds a separate, standalone UI component which communicates via the text interface with the proof assistant. It is this architecture which will be assumed in the sequel by default. This is

of course the architecture underlying systems such as ProofGeneral, PVS and PCoq.

Does it make sense to go even further and divide the UI into separate standalone components? For example, what about using a standard web browser for reading theories, a self-contained component for specifying theories and a separate interactive proof component for deriving new theorems? Here are a couple of considerations that need to be taken into account:

- In general, building a monolithic environment with little reuse can bring performance improvements and does not suffer from integration problems. However, it is usually infeasible because of a prohibitive amount of extra work compared to the reuse of existing components.
- Splitting off a browsing component is in a way easier than splitting off one of the other two components. This is because the browsing component does not affect the other two - there are no mutual dependencies. Synchronisation only requires an update of the data/documents which is affected by activities of the editor and the proof component.
- Using a standard web browser for reading theories is attractive because of reuse possibilities and widespread familiarity with the interface. MathML promises to simplify the rendering of mathematical formulas. Representing theories as XML based documents allows simple customisation of the presentation via XSLT resp. via a transformation to HTML and the use of CSS style sheets.
- Splitting apart theory specification and proof derivation components can lead to problems with concurrent access. In particular, a non-conservative theory operation can invalidate concurrent proof attempts. There are also issues relating to the architecture of the proof assistant. For example, for the HOL family of provers, the components would need to talk to a single theorem prover process. Thus a concurrent use of such components would have to be interleaved.

Proof Assistants as Components

According to the architecture suggested above, the proof assistant is not used directly via a console interface but indirectly via a user interface. This amounts to a significant change for some proof assistants such as those from the HOL/LCF family. Traditionally, a system such as Isabelle has been viewed as complete in itself, rather than as being a component of a larger system. This change of paradigm has as number of implications for the proof assistant. For example, if proof scripts are generated automatically from a "proof by clicking", then it is no longer paramount for proof commands to be in a terse human-oriented syntax. Instead, for the purposes of communicating with the UI, either (remote) procedure calls or messages in an XML based protocol are more appropriate.

More generally, the separation of users interfaces from the rest of the system invites a view of proof assistants as reactive components that provide a mathematical service in a distributed environment. This raises many questions, ranging

from security issues and the support for concurrent access to the question of how to deal with a prover that does not respond due to an infinite loop in a proof attempt.

Allocation of Responsibilities

A basic system design problem is the allocation of responsibility between different parts. In case of a UI and a proof assistant backend, the assignment of most tasks is not a real question - the roles of the two components are clearly defined and quite separate. However there are a couple of points worth discussing:

- For efficiency reasons, it is advisable that the UI performs basic validation of user inputs.
- Traditionally, parsing and pretty-printing are allocated to the proof assistant. Assuming an XML based communication protocol between the two subsystems (see below), this distribution of work will need to be reviewed. According to the general principle that data should be transformed into a "presentation form" as late as possible, it is advisable to do the final "pretty-printing" transformations in the UI component. In case of parsing textual input by the user, the case is not so clear cut as such a parser might be needed in the proof assistant anyway in order to process proof script files. There seem to be a number of reasonable options, including simply wrapping up of textual inputs in XML or the sharing of a separate parser component between UI and the proof assistant.
- For efficiency reasons, one might consider to replicate part of the proof assistant state in the UI. For example, by replicating the history mechanism, one can reduce the amount of traffic when undo-ing a proof step as the new proof state can be determined locally in the UI. However, the simpler design is to keep these responsibilities with the proof assistant as it is the "expert" as far as the proof state and logical contexts are concerned. The final decision about replication depends to a large extend on the anticipated deployment scenario. For example, if the UI and the proof assistant are running on the same machine, then it it doubtful that the performance gained justifies a more complex design.

Component Frameworks and XML Protocols

Object-based component frameworks [25] such as CORBA, COM+ and Enterprise Java Beans (EJBs) support distributed computing based on the declaration of interfaces and methods. They rely on quite extensive runtime environments that hide many of the complexities of programming in a distributed and/or heterogeneous environment. In addition these frameworks also provide a varying amount of support for other "middleware services" such as transactions, security, database pooling, pooling of object instances, etc. In these frameworks, components are associated with one or more interfaces consisting of typed methods that can be invoked from other components. The framework ensures type safety,

i.e. the methods in a component implementation have to comply with the types (including possibly exceptions) in the interface declaration for that component.

For an application, the best choice of component/distributed computing framework depends in general on many factors including the amount of distribution and concurrency as well as the implementation languages and the target operating systems. Having said this, it would appear that because of the platform diversity in the academic community, a single operating system solution such as COM+ is currently hardly acceptable as a framework for building reusable UI and theorem prover components. It is also not clear that many of the facilities offered by CORBA, COM+ and EJBs would be useful with the current generation of proof assistants. Because of the non-neglectable overhead, the chosen framework should be as simple as possible. In light of these arguments, it seems that simpler approaches are needed. For platforms written in Java, the PCoq approach of encapsulating foreign language components as Java processes seems a natural framework. Components are accessed via Java APIs and distribution can be supported using either sockets or Java RMI.

Recently, XML message passing has appeared as an alternative approach to distributed computing. It has attracted a lot attention in form of SOAP or XML-RPC based web services that run via HTTP [26]. In this approach, components provide typically only a small number of interfaces over which data encoded as XML documents are exchanged. The validity of XML messages with respect to pre-defined rules is not ensured by the framework but (if at all) is instead checked by the receiver of the message. This results in very loosely coupled, easily extensible and lightweight architectures. XML processing and web service standards are supported by implementations for different platforms and programming languages while at the same time being completely platform and programming language independent. These features make XML messaging an attractive choice for the communication between generic UIs and proof assistant backends.

D. Aspinall has written a white paper [27] that suggests an XML vocabulary for communication between the major components in a theorem proving environment. Here are a couple of general remarks on such an XML based protocol between UI and proof assistant backend.

– The protocol needs to specify both the set of permissable XML messages (the "vocabulary") as well as rules about the permissable sequencing of messages; in particular what are the allowed request/response pairs.
– Following the already mentioned principle of keeping data in its semantic form as long as possible, it is suggested that the markup of proof assistant responses should be semantic rather than presentation-oriented.
– For performance reasons, it is essential that the protocol supports coarse-grained concurrency. This means that the proof assistant backend can process large units in one chunk without the need for interim synchronisation. For example, it is desirable to have XML markup for a "process theory XYZ" request such that no other messages are exchanged until the theory either is completely loaded or a failure occurs.

- The XML vocabulary should only regulate aspects which are directly relevant for the UI/prover communication and which are not covered by other standards such as MathML.
- The XML vocabulary needs to be planned with future extensions in mind.
- In order to further reuse, the XML protocol should contain a common core that is independent of proof assistants and user interfaces.

It is clear that the last point might well prove difficult in practice - getting different proof assistant and UI implementors to agree on a standardised core XML vocabulary and message sequencing rules will not be easy.

Adaptability

Following the platform paradigm, the suggested theorem proving environment features an extensible set of components such as theory editors, theory browsers, interactive proof components, MathML display engines, connectors to other provers, etc. According to the principles of component-based software engineering, these software parts should be easily reusable and customisable. In order to achieve this, one should aim for relatively generic interfaces during the design. This should be followed by implementations that follow the principle of "programming to interfaces". The formulation of generic interfaces can benefit from considering different application scenarios. For example one might want to make it an explicit requirement that a proof display component supports two different provers.

In practice, a simple "plugging in" of more advanced components such as a tree editor can not be expected. Even if there were standardised interfaces for such components, a specific component implementation might well provide additional features requiring adapter code. However for such complex components, this is justified. The situation is different for simpler parts such as XSLT input/output filters. In this case, the platform should provide standard hooks for plugging in such a component. For instance, a program verification project could use a specialised filter that transforms programs from their usual syntax into an abstract syntax tree representation processed by the proof assistant. The addition of such a component to the platform should be possible without having to add program code.

For theory proving applications, the dynamic extension of a running system with new components does not seem necessary. Instead it seems sufficient to use configuration files that are loaded at startup time. This is the approach taken in the XUL [28] framework which underlies recent versions of the Mozilla web browser suite. As this project shows, it is quite feasible to extensively adapt user interfaces with the help of configuration files.

7 Concluding Remarks

Proof assistants will only realise their full potential once they are accompanied by effective user interfaces. It is hoped that the reflections on requirements and design in this paper will contribute to the achievement of this goal.

Despite the aim to stay generic, the discussion has doubtlessly been influenced by the author's experience with Higher-Order Logic (HOL) proof assistants. This bias should be rectified in future work by looking more closely at other automated reasoning systems and their user interfaces.

Many issues have not been touched at all. This is particularly regrettable in case of proof planning [29], multi-modal facilities [10], recent projects that use the web to distribute mathematical knowledge [30], and cooperative theorem proving [17, 4].

References

1. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V7.4. (2003) http://coq.inria.fr.
2. Gordon, M.J.C., Melham, T.F., eds.: Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press (1993)
3. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
4. Allen, Constable, Eaton, Kreitz, Lorigo: The Nuprl open logical environment. In: CADE: International Conference on Automated Deduction. (2000)
5. Siekmann, J., et al.: Proof development with OMEGA. In Voronkov, A., ed.: Automated Deduction – CADE-18. LNCS 2392, Springer-Verlag (2002) 144–149
6. S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringert-Calvert: PVS System Guide – Version 2.4. (2001) http://pvs.csl.sri.com.
7. Wiedijk, F.: Comparing mathematical provers. In: Mathematical Knowledge Management, Proceedings of MKM 2003. (2003) 188–202
8. David Aspinall and Thomas Kleymann: Proof General User Manual – Version 3.4. (2002) http://www.proofgeneral.org.
9. Amerkad, A., Bertot, Y., Rideau, L., Pottier, L.: Mathematics and proof presentation in Pcoq. In: Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01). (2001) http://www-sop.inria.fr/lemme/pcoq/.
10. Siekmann, J., Hess, S.M., Benzmüller, C., Cheikhrouhou, L., Fiedler, A., Horacek, H., Kohlhase, M., Konrad, K., Meier, A., Melis, E., Pollet, M., Sorge, V.: $\mathcal{L}\Omega\mathcal{UI}$: $\mathcal{L}$ovely Ω $\mathcal{U}$ser $\mathcal{I}$nterface. Formal Aspects of Computing **11** (1999) 326–342
11. Lüth, C., Wolff, B.: More about TAS and IsaWin: Tools for formal program development. In Maibaum, T., ed.: Fundamental Approaches to Software Engineering FASE 2000/ ETAPS 2000. LNCS 1783, Springer Verlag (2000) 367–370
12. W.M. Farmer, J.D. Guttman, F.J. Thayer: The IMPS User's Manual First Edition, Version 2. (1995) http://imps.mcmaster.ca/.
13. Lüth, C., Wolff, B.: sml_tk: Functional programming for GUIs – reference manual. Technical Report 158, Albert-Ludwigs-Universität Freiburg (2001)
14. Xavier Leroy: The Objective Caml system release 3.06 – Documentation and user's manual. (2003) http://www.ocaml.org.
15. Jacobson, I.: Object-Oriented Software Engineering – A use-case driven approach. Addison-Wesley (1992)
16. Larman, C.: Applying UML and Patterns. Prentice Hall PTR (2002)
17. Goguen, J.: Social and semiotic analyses for theorem prover user interface design. Formal Aspects of Computing **11** (1999) 272–301
18. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1999)

19. Meyer, B.: Object-Oriented Software Construction. second edn. Prentice-Hall, Englewood Cliffs (1997)
20. Dix, A., Finlay, J., Abowd, G., Beale, R.: Human-Computer Interaction. Prentice Hall (1998)
21. Eastaughffe, K.: Support for interactive theorem proving: Some design principles and their application. In Backhouse, R., ed.: Proceedings of the Workshop on User Interfaces for Theorem Provers. Computing Science Reports, Eindhoven. (1998)
22. Huisman, M.: Reasoning about Java Programs in higher order logic with PVS and Isabelle. Ipa dissertation series, 2001-03, University of Nijmegen, Holland (2001)
23. Merriam, N., Harrison, M.: What is wrong with GUIs for theorem provers (1997)
24. Bertot, Y., Thery, L.: A generic approach to building user interfaces for theorem provers. Journal of Symbolic Computation **25** (1998) 161–194
25. Heinemann, G.T., Councill, W.T.: Component-Based Software Engineering: putting the pieces together. Addison-Wesley (2001)
26. Cerami, E.: Web Services Essential. O'Reilly (2002)
27. David Aspinall: Proof General Kit – White Paper (Version 1.4). (2000) http://www.proofgeneral.org/kit.
28. Bullard, V., Smith, K., Daconta, M.: Essential XUL Programming. Wiley (2001)
29. Bundy, A.: A critique of proof planning. In andFariba Sadri, A.C.K., ed.: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. Volume 2408 of LNCS., Springer (2002) 160–177
30. Kohlhase, M., Zimmer, J.: System description: The MathWeb software bus for distributed mathematical reasoning. In Voronkov, A., ed.: Automated Deduction – CADE-18. LNCS 2392, Springer-Verlag (2002) 139–143

Author Index

Aspinall, David, 1
Audebaud, Philippe, 14
Autexier, Serge, 81

Benzmüller, Christoph, 81
Bertot, Yves, 32
Bornat, Richard, 46, 62

Giese, Martin, 74
Guilhot, Frédérique, 32

Hübner, Malte, 81

Kiniry, Joseph, 101

Lüth, Christoph, 1

Meier, Andreas, 81, 133
Melis, Erica, 133

Owens, Scott, 123
Owre, Sam, 101

Pollet, Martin, 133
Pottier, Loïc, 32

Rideau,Laurence, 14

Slind, Konrad, 123

Théry, Laurent, 143

Völker, Norbert, 160