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.

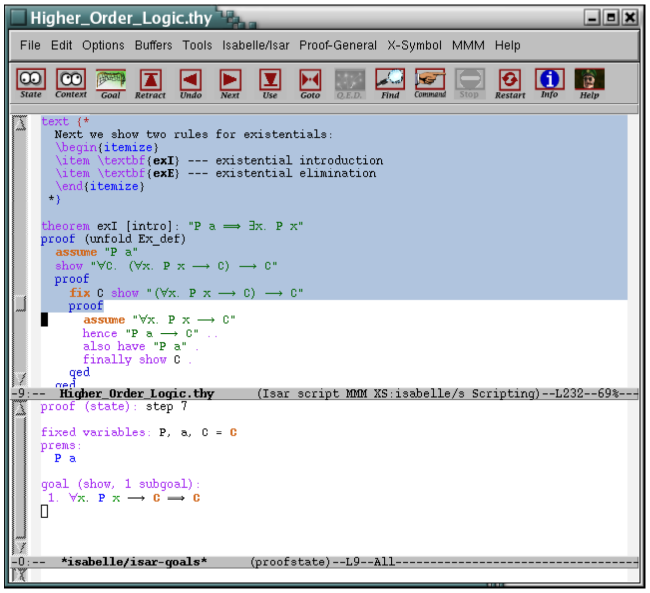
In the remainder of this paper, we describe the particular aspects of Proof General and IsaWin that we want to build on, followed by a outline design of the new system. We close by briefly mentioning related work, and our vision for the future evolution of the project.

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.


Higher_Order_Logic.thy
File Edit Options Buffers Tools Isabelle/Isar Proof-General X-Symbol MMM Help
State Context Goal Retract Undo Next Use Goto Q.E.D. Find Command Stop Restart Info Help

```
text {*
  Next we show two rules for existentials:
  \begin{itemize}
  \item \textbf{exI} --- existential introduction
  \item \textbf{exE} --- existential elimination
  \end{itemize}
 *}

theorem exI [intro]: "P a ⟹ ∃x. P x"
proof (unfold Ex_def)
  assume "P a"
  show "∀C. (∀x. P x ⟶ C) ⟶ C"
  proof
    fix C show "(∀x. P x ⟶ C) ⟶ C"
    proof
      assume "∀x. P x ⟶ C"
      hence "P a ⟶ C" ..
      also have "P a" .
      finally show C .
    qed
  qed
-9:--  Higher_Order_Logic.thy      (Isar script MMM XS:isabelle/s Scripting)--L232--69%---
proof (state): step 7

fixed variables: P, a, C = C
prems:
  P a

goal (show, 1 subgoal):
 1. ∀x. P x ⟶ C ⟹ C

-0:--  *isabelle/isar-goals*      (proofstate)--L9--All-----------------------------------
```

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

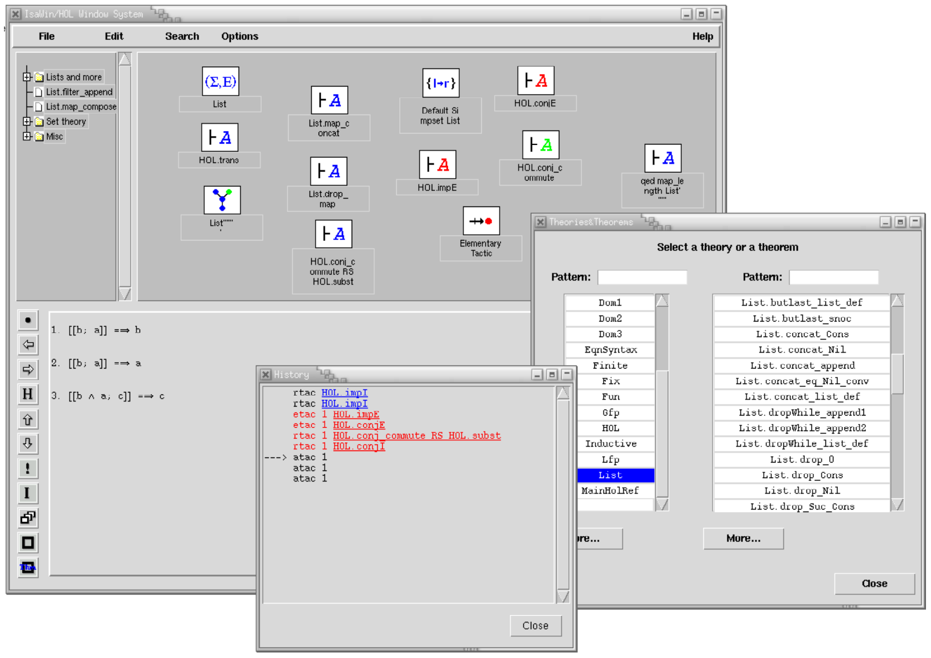
To address the limits of the Proof General model, and particularly of the Emacs Lisp implementation, Proof General (PG) Kit has been designed [4, 5]. The central idea is to use the experience of connecting to diverse provers to prescribe a uniform protocol for interaction. Instead of tediously adapting 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.



```
1. [[b; a]] ==> b

2. [[b; a]] ==> a

3. [[b ∧ a; c]] ==> c
```

```
     rtac HOL.impI
     rtac HOL.impI
     etac 1 HOL.impE
     etac 1 HOL.conjE
     rtac 1 HOL.conj_commute RS HOL.subst
     rtac 1 HOL.conjI
---> atac 1
     atac 1
     atac 1
```

```
Dom1
Dom2
Dom3
EqnSyntax
Finite
Fix
Fun
Gfp
HOL
Inductive
Lfp
List
MainHolRef
```

```
List.butlast_list_def
List.butlast_snoc
List.concat_Cons
List.concat_Nil
List.concat_append
List.concat_eq_Nil_conv
List.concat_list_def
List.dropWhile_append1
List.dropWhile_append2
List.dropWhile_list_def
List.drop_0
List.drop_Cons
List.drop_Nil
List.drop_Suc_Cons
```

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]

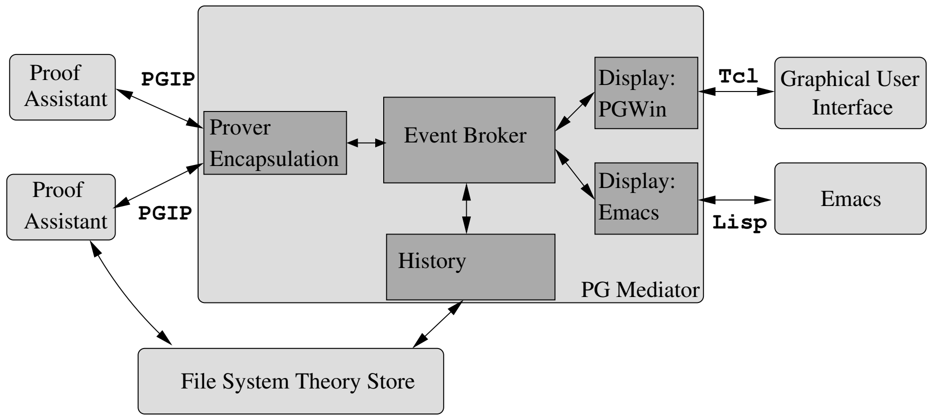


Fig. 3. System Architecture

[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. *Ω*Mega: 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.