Proof General / Eclipse: A Generic Interface for Interactive Proof

Daniel Winterstein[1], David Aspinall[1], and Christoph Lüth[2]

[1] LFCS, School of Informatics, The University of Edinburgh, U.K.
[2] Department of Mathematics and Computer Science, Universität Bremen, Germany

Abstract. Interactive theorem provers are becoming increasingly powerful and sophisticated. However user-level support for interactive theorem provers is often wanting, and interaction can be frustratingly low-level.
This paper introduces PG/Eclipse; a sophisticated new interface for interactive theorem provers, offering users a rich set of proof development tools. These include *script management* (a form of step debugging tailored to linear scripts), and theory browsing tools. It is based upon the PG/Kit, a generic framework that allows straightforward adaptation to most interactive theorem provers. Moreover, by separating interface development from proof engine development, this framework should facilitate the development of both. PG/Eclipse is built upon the highly modular and extensible Eclipse toolkit, making it a good platform for interface research.

1 Introduction

Developing formal proofs is an arduous and difficult task. Nevertheless, larger and more complex formalisations are being undertaken in numerous interactive theorem proving systems ("provers" for short). However inspite of the considerable achievements of the formal proof programme, take-up of these systems by mathematicians and programmers remains poor. At least one reason for this is the lack of good development tools.

For most of these systems, the record of instructions of how to create the proof, is kept in a text file with a programming language style syntax. We call these files *proof scripts*. For large examples, the proof scripts are also large: e.g. work on formalising Java in `Isabelle` (30k lines, 1400 lemmas [18]) or the Fundamental Theorem of Algebra in `Coq` (80k lines, almost 3000 lemmas [9]). Each of these cases represents several person-years of work.

Yet the facilities for developing and maintaining formal proofs are in general rather rudimentary, especially when compared with the sophisticated IDEs available to the modern programmer. One reason for this is the fragmentation of the community of formal proof developers across different systems, each using its own language and logic, which means that the users of each system do not have the resources to provide sophisticated tools.

1.1 PG/Emacs

The *Proof General* (PG) project is an ongoing attempt to redress this issue [3, 4]. The previous PG interface was built on the Emacs text editor (we will refer to it here as PG/EMACS to distinguish it from the new interface). PG/EMACS has been successfully used for several years, with an estimated ?? users. Its success is due to its genericity, allowing easy adaption to a variety of provers (primarily, `Coq`, `PhoX`, and `Isabelle`, for which it is the 'official' interface), and its design strategy, which targets both experts and novice users.

Although successful, the limitations of the PG/EMACS system are becoming increasingly clear. From the users' point of view, it requires learning Emacs and putting up with its idiosyncratic and at times unintuitive UI. From the developers' point of view, it is rather too closely tied with the Emacs Lisp API which is restricted, somewhat unreliable, often changing, and differs between different flavours of Emacs.

Another engineering disadvantage of PG/EMACS arose from its construction by successively extending a generic basis to handle more provers. This strategy meant that little or no specific adjustment of the provers was required, but it resulted in overcomplicated configuration and internal mechanisms.

1.2 PG/Eclipse

In this paper we present PG/ECLIPSE, an interface/development environment that marks a new phase in the PG project. It not only provides a more sophisticated suite of editing, browsing and debugging tools – it does so using a generic framework that should allow it to be used by a wide range of systems. We first present the features it offers. We then describe the framework it operates in (called the PG/KIT), which is based around a protocol for handling interactive proof sessions (called *Proof General Interactive Protocol* or PGIP), and give an overview of the implementation.

2 The PG/Eclipse interface: a functional description

This section presents the main features of PG/ECLIPSE from a user-perspective. Many of these features will be familiar to users of modern programming IDEs, where they are now increasingly standard. Functionality can be accessed via buttons and menus for new users, and via key-sequences for experienced users.

2.1 Script management

The central feature of PG/ECLIPSE is an advanced version of *script management* (c.f. [**?**]), which is a form of step debugging tailored to linear scripts.

To explain script management, consider the short example proof script in Fig. 1. To interactively "run" this script, we sequentially send each line to the prover. Script management says that each script can be divided into a part which

```
lemma fn1: "(EX x. P (f x)) --> (EX y. P y)"
proof
  assume "EX x. P (f x)"
  thus "EX y. P y"
  proof
    fix a
    assume "P (f a)"
    show ?thesis ..
  qed
qed
```

Fig. 1. An example proof script in Isabelle/Isar.

has already been processed, a part which is currently being processed, and a part which has not been processed (yet). Proof General supports this by colouring the parts of the text being processed or already processed, and preventing editing in those regions.[3] This provides clear feedback to the user on processing, and protects against edits that might invalidate the current proof state. A toolbar provides buttons for navigating (moving the prover's position) within the proof. An attractive feature of the interface is that these navigation buttons behave identically across numerous different systems despite the fact that behind the scenes, the provers can be very different. Moreover the user does not have to know the low-level commands needed for navigation. This is reminiscent of the way a web browser provides a uniform metaphor for browsing across file: and ftp: as well as http: URLs.

A screenshot in Fig. 3 shows this in action. The main editor window shows the proof script; view windows below show the prover output. An Eclipse Problems view (not shown here) lists outstanding problems, such as syntax errors or unfinished proof-goals. To the left of the editor window is an Outline view of the proof script showing its structure. Above the editor, the dedicated toolbar triggers proof or undo steps by sending instructions to the prover (left-to-right, the buttons are: undo all, undo, interrupt, step forward, process all, go to cursor location).

2.2 Symbol support

Using mathematical symbols can make a huge difference to how readable a script is (e.g. compare reading "`\forall x, y . x \and y \implies x`" with "$\forall x, y. x \wedge y \Rightarrow x$"). PG/ECLIPSE provides symbol support similar to the Emacs XSymbol package [19]. Like XSymbol, it supports the use of typing shortcuts to enter symbols (e.g. typing "- ->" for "\<longrightarrow>"). At present, it is slightly less powerful than XSymbol in that it does not support subscripts and superscripts. PG/ECLIPSE also provides a symbol table editor (shown in Fig. **??**), allowing users to adapt and extend the use of symbols to fit their own needs.

[3] The colouring of script management is demonstrated in the screenshot in Fig. 3.

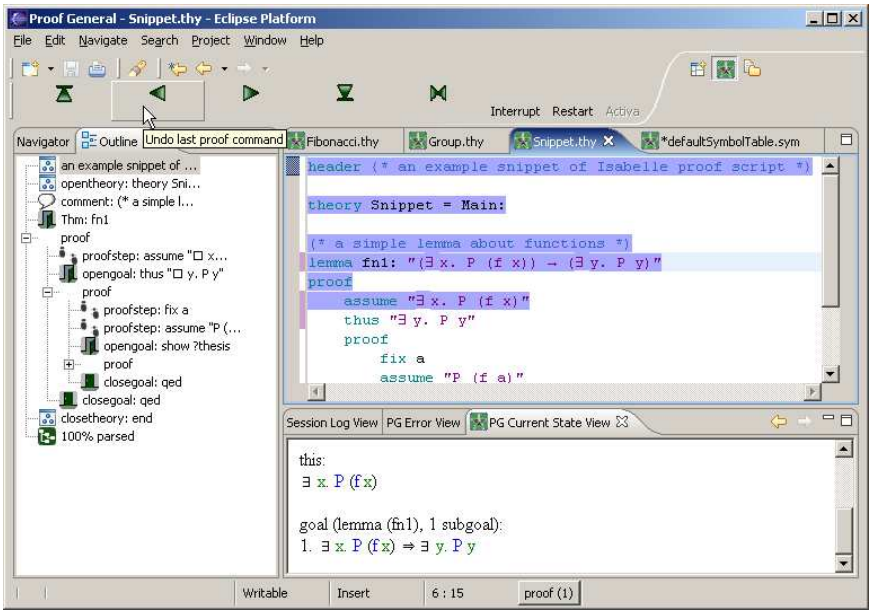

Fig. 2. Eclipse Proof General Display

Symbol support is based on Unicode, which has the advantage that the user can cut and paste text between modern applications, preserving symbols. The disadvantage is that the symbol support available depends on access to a suitably rich unicode font.[4]

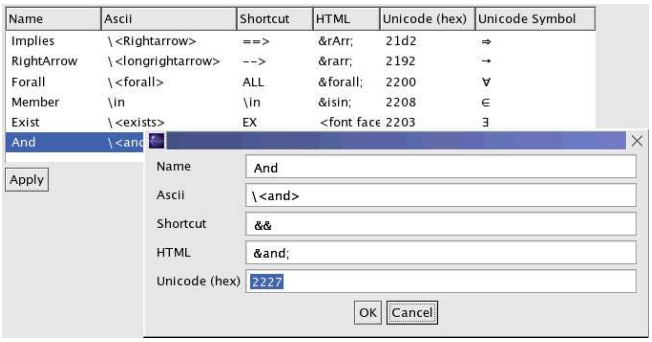

Fig. 3. The PG/ECLIPSE Symbol-Table Editor

[4] Standard Linux and Windows installations should be fine; we have not tested on Macs.

2.3 Theory navigation

As a theory grows, finding specific definitions and proofs can become increasingly difficult and time consuming. This alone can make a large difference to the usefulness of a theory. PG/ECLIPSE provides support for theory navigation. As proof script files are read, their structure is analysed and the theories, theorems and lemmas they contain are indexed. This allows the user more ways to navigate a project:

- Via a 'Theory Index View' (which supports user-specified filters). This is a list of previously processed theories, theorems and definitions. The list entries are hyperlinked to the relevant proof-script section, allowing quick access.
- Via a similar 'Problems View' shows a (hyperlinked) list of unsolved problems, such as unproved goals or syntax errors.
- Via a 'jump to definition' command, that allows users to look-up the source for a rule.
  Within a file, a zoomable/collapsible outline view shows the file structure at a glance and allows quick navigation. There are also Eclipse's in-built indexing and browsing features, such as 'edit browsing'(which takes the user back through the locations of recent edits in a file), book-marks and To-Do lists.

2.4 Content Assistant

The Content Assistant suggests completions for keywords/phrases. It is like the Auto-Completion feature found in word-processors, but activated by a hot-key combination rather than by typing alone (making it less intrusive). At present, it uses a list of theorems plus a static list of keywords. However there is support in PGIP for the prover to amend this list and provide more context sensitive suggestions.

2.5 Integrated help

As with programming, good documentation is key to both the uptake of theorem provers and the reuse of proof scripts. PG/ECLIPSE provides tools for documenting prover commands, prover settings and proof scripts.

Help for prover commands can be defined via a simple XML file. Help for prover settings (e.g. heap size, start-up directory, etc.) is handled similarly. PG/ECLIPSE provides a GUI front-end for viewing and changing prover settings. This is configured via another simple XML file, and can include help on these settings. Help for theory elements (e.g. theorems) is created just by writing a preceding comment. This is based on the JavaDoc approach to documenting code (c.f. [**?**]). It gives a method for documenting proof scripts whilst writing them, and moreover one which is straightforward (and will already be familiar to many users), and not overly time-consuming.

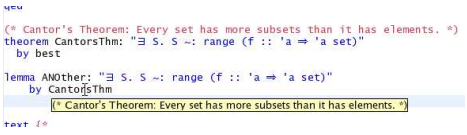

Fig. 4. Providing focused help via tooltips.

In all of these cases, help is then presented to the user via tooltips shown in response to a pause or 'hover' by the user (see Fig. 4).

Help on the PG/ECLIPSE system itself is provided via links to a Wiki[5]. This is, we hope, a good solution to the problem of documenting systems that serve a small community (i.e. systems where the developers cannot afford to provide professional level support and documentation). The PG Wiki includes a mechanism for requesting help from the developers (text of the form "QUESTION: blah blah blah" is interpreted as a request for help). This has the benefits of a Q&A user-group, whilst simultaneously creating structured documentation. It also allows users to easily share information, thus alleviating some of the documentation burden on the developers. Although PG/ECLIPSE's current documentation is slight, using a Wiki means that it can grow as necessary in response to user demands.

2.6 Teaching tool

One of the great potentials for provers, & as yet largely untapped, is in mathematical/scientific education. Although there are excellent computer algebra and graphing systems used in education (e.g. Maple [?]), these neither perform nor teach proofs, which is central to mathematical work.

PG/ECLIPSE introduces a teaching tool, designed for delivering teaching material that interacts with a prover.[6] The tool uses an embedded browser to display web-pages, and defines some Javascript commands for interacting with the script editor (e.g. opening a proof-script containing exercises). The use of a web-browser means this tool is flexible, and should be easy to integrate with existing web-based teaching material.

2.7 Interface scripting

Often a script will run better or view better under certain settings. PG/ECLIPSE allows a theory developer to encode these settings in the proof script file. It defines a small set of commands that can be used to script the interface itself. Currently, there are commands for extending symbol support (e.g a script can specify that `myOperation` should display as $\star$), and for setting preferences, both of the interface and of the underlying prover. These commands can be embedded

[5] A *Wiki* is a website that can be edited by any visitor.
[6] This tool could also be used to deliver documentation linked to 'live' examples.

inside comments, so that the proof scripts will still run outside of PG/Eclipse.[7] This feature could also be useful when using PG/Eclipse as an educational tool (where, for example, some features might want to be temporarily disabled).

3 Proof General Kit architecture

The claim that we can provide a uniform framework for interactive proof seems bold. Why can we expect to provide useful tools at a generic level for a dozen different provers? Especially when those provers do not just differ in their underlying languages, but also in their existing interaction mechanisms as well.

Our solution has been to apply the experience gained from the PG/Emacs project in designing a new protocol-based approach. It is unrealistic to expect that a prover should not need modification to support a sophisticated interface. So instead of trying to match a range of different behaviours, we propose a uniform API which captures the behaviour common to most provers at an abstract level, and ask that each proof system implements that. If done correctly, this will not place a great burden on the prover developer, and by defining a clear separation between the interface and the prover, it should greatly facilitate interface development.

The PG/Kit API consists of two linked parts: A model for prover behaviour, and a protocol for communication within proof sessions. This design reflects a compromise between creating a flexible system, and working with existing systems. An alternative would be to use a more communication based approach, where the prover informs the interface about the proof-script relevant effects of executing commands. For example, consider the `undo` command. the PG/Kit model specifies how undo behaviour should change depending on context; the communication-based approach would be to leave undo behaviour unspecified, and for the response to an undo command to specify what has been undone. This would be more generic, but would require a greater change from existing systems.

3.1 Modelling the prover state

PG/Kit assumes an abstract model of how interactive provers behave, where we suppose there are four fundamental states occupied by the prover. Transitions between the states are triggered by commands issued via the interface. Fig. 5 shows the states, and the commands to change between them. The four states illustrated are:

1. the *top level* state where nothing is open yet;
2. the *file open* state where a file is currently being processed;
3. the *theory open* state where a theory is being built;
4. the *proof open* state where a proof is currently in progress.

[7] This is similar to the way that Javascript used to be hidden inside html comments, so that the document would render correctly on old browsers.

These states correspond to a hierarchy of named items: The top level may contain a number of files. A file contains a proof script, structured into theories, and theories in turn may contain theory items (constant declarations etc.) and proofs consisting of proofsteps.

The reason for distinguishing the states is that different commands are available in each state, and the prover's undo behaviour in each state can be different.

This model is based on abstracting the common behaviour of many interactive proof systems, but it is not intended to capture precisely the way every proof system works. Rather it acts as a clearly specified "virtual layer" that must be emulated in each prover to cooperate properly with the broker.

3.2 PGIP: A protocol for interactive proof

The protocol for directing proof used by PG/KIT is called *PGIP*, for *Proof General Interactive Proof* [5]. It was designed by examining the communications used in PG/EMACS, and covers a wide range of prover-display interactions. The format of the messages is defined by an XML schema (written in RELAX NG [13], but also available as an XML DTD). Messages are sent over channels (both sockets and Unix pipes are supported). It is not necessary for a prover to support all of PGIP in order to use the PG/ECLIPSE interface (and indeed, at present PG/ECLIPSE does not implement all of PGIP). We distinguish several different types of command in PGIP:

Proof script commands correspond to the commands in a conventional proof script, and would typically be created from such a script. They affect the internal (proof-relevant) state of the prover.

Improper commands are those which should not appear in a proof script. They are used for controlling the proof session; examples are the *undo* and *abort* commands appearing in Fig. 5 below.

Display messages are sent from the prover, and contain output directed to the user, such as the current proof state or error messages.

Parsing commands define a mechanism whereby PGIP mark-up is added to sections of 'raw' proof scripts (i.e. scripts in the language of the prover).

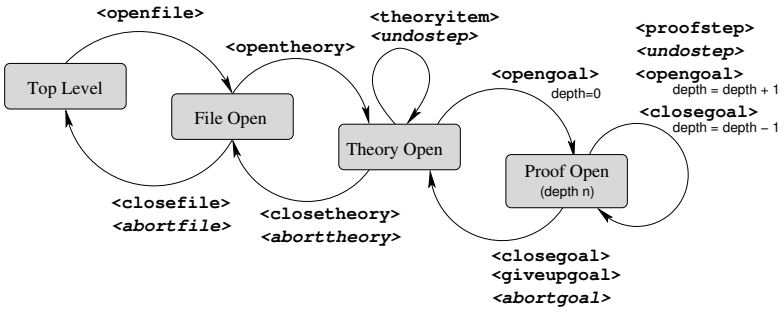

Fig. 5. Proof states during development.

We expect that this task will normally be implemented as part of the theorem prover, which should know how to parse its own language.

Configuration messages are used for initially setting up components.

Proof state descriptions describe objects in the current proof environment; e.g., the theorems that are currently available.

Other message kinds include system inspection and control commands, and meta data sent from the prover, for example, dependency information between loaded files and proven theorems.

3.3 Proof scripts in PGIP

Proof scripts are the central artefact of the system.

The basic principle for representing proof scripts in PGIP is to use the prover's native language, and mark up the content with PGIP commands which give the proof script the structure needed by an interface. For example, Fig. 6 shows the PGIP representation of the example proof script from Fig. 1 with the structural markup. Notice the named and unnamed <opengoal> elements, and the indentation structure introduced by <opengoal> and <closegoal>.[8]

```
<opengoal name="fn1">
    lemma fn1: &quot;(EX x. P (f x)) --&gt; (EX y. P y)&quot;
</opengoal>
<proofstep>proof</proofstep>
  <proofstep>assume &quot;EX x. P (f x)&quot;</proofstep>
  <opengoal>thus &quot;\<exists> y. P y&quot;</opengoal>
    <proofstep>proof</proofstep>
    <proofstep>fix a</proofstep>
    <proofstep>assume &quot;P (f a)&quot;</proofstep>
    <opengoal>show ?thesis</opengoal><closegoal>..</closegoal>
  <closegoal>qed</closegoal>
<closegoal>qed</closegoal>
```

Fig. 6. A proof script in Isabelle/Isar, marked up in PGIP.

This design decision has two main consequences:

Firstly, the user employs the prover's native language (such as Isar above) to write the proof scripts instead of one generic language. This is both necessary, since the wide variety of logics and proof styles supported by modern provers make it very hard to come up with one language which efficiently supports all of these, and pragmatic; users can continue to work in the proof language they are

[8] One may wonder why <opengoal> and <closegoal> are separate and distinct elements; we do not use a single <goal> element to enclose the block structure because we need to be able to incrementally parse and evaluate text, which means handling ill-structured fragments of a block.

familiar with and old proof scripts can still be used. Moreover, the interface does *not* lock the user into always using it. Proof scripts developed with PG/ECLIPSE can be run independently from the interface.

Secondly, the parsing of the proof script, and thus all user input, has to be done by the prover (or a component coming with the prover), and not by the interface, which knows nothing about the prover's language. Note that this parser only needs to understand *some* aspects of the prover's language – principally, it must be able to distinguish between comments, normal proof commands, commands that open a new goal, or close an open goal, and 'forgettable' commands that do not need to be undone (c.f. Fig. 5, where the command types that must be identified (except for comment) are shown as labels on the state transition arcs). It should not be arduous to develop such parsers (although c.f. §4.4 for a case study).

4 Implementation

4.1 Using Eclipse

The restrictions of Emacs have led us to adopt Eclipse as the new platform for developing Proof General. The new interface is designed to replace and extend the current Emacs-based interface, with the goal of producing a proof development environment akin to the sophisticated modern IDEs available for programming. Eclipse provides the ideal platform for this.

Eclipse is an open-source IDE and tool integration platform written in Java and SWT (SWT is a widget set created by IBM, which exploits native operating system widgets to provide an interface that is both faster and more familiar than most cross-platform GUIs) [15, 8]. Most prominently, Eclipse provides a powerful and attractive IDE for Java, but it also has a modular design based on *plugins* and *extension points* that allows almost any aspect of the platform to be customised and extended to new domains. Many plugins are available, supporting other programming languages, profiling and testing tools, graphical modelling facilities, etc.

Amongst the features Eclipse offers are:

- A state-of-the-art user-friendly GUI.
- Integrated CVS support, including a CVS client with a GUI and a comparison editor for accepting/rejecting differences.
- Good platform independence. PG/ECLIPSE can be installed on Linux, Windows or Mac boxes without any special configuration. Of course, the underlying theorem prover may be platform dependent, but a socket layer allows the editor to be run on a different machine from the prover.

Extending Eclipse is not as straightforward as one might hope. To provide functionality similar to that offered for Java, it is necessary to re-implement much of the functionality. The learning curve for using Eclipse is quite steep. One of Eclipse's aims is to provide an extensible API that is cleanly separated

from the code-base, with custom extensions created using XML schemas rather than via Java code. In practice, this has only partially been achieved. Most extensions require Java code to complete their setup, which means engaging with a complex and inter-linked code-base. For example, to set up basic syntax highlighting (e.g. keyword and comment colouring – something that can be done with one straightforward file in many editors), requires an XML configuration file plus properly sub-classing and setting up six different Java classes.

However, Eclipse does provide a powerful and wide-ranging set of tools, plus considerable support for creating new tools that work within a development environment. These tools are also frequently interlinked, so that implementing one aspect of a tool can unlock a range of extra functionality 'for free'. Also, Eclipse imposes relatively few limits on what can be implemented within its framework.

The PG/ECLIPSE plugin is implemented with around 10 Java packages containing about 100 classes, most of which contain small pieces of code to interface to the Eclipse platform. It represents roughly one year's work.

4.2 PG/Eclipse Architecture

The architecture of the main editing loop of PG/ECLIPSE is shown in Fig. 7. PG/ECLIPSE is implemented along Model-View-Controller lines. On loading, proof scripts are converted into structured 'document objects' – this is the model. This is presented to the user via a specialised text-editor and associated viewers (e.g. the document outline view). These 'views' support various actions (the 1controller' aspect), most importantly text editing and script management.

Communication with the prover is handled via the ??PGIP Gateway module, which converts between internal command and event objects, and PGIP messages. This module maintains a model of the prover's state, which is necessary to properly implement actions such as `undo`.

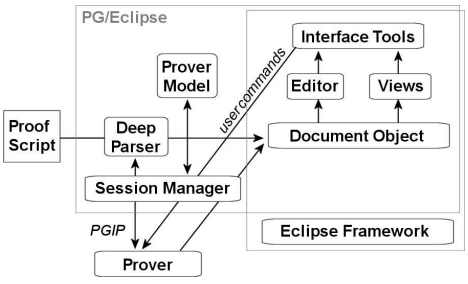

Fig. 7. PG/ECLIPSE system architecture.

There are also several parts of PG/ECLIPSE that sit outside the editing loop, and are not shown in 7. Most important of these are the views giving feedback

on the session: the Current State View, an error log view, and a session log view. These views simply listen to the event traffic generated by the PGIP Gateway, and display appropriate portions of it. Other aspects of PG/ECLIPSE not illustrated here are the symbol table editor, the teaching tool, and the preference system.

4.3 Parsing

The Eclipse editor window must deal with managing information gleaned from the structure of the script, while allowing free form text edits – which can wreak arbitrary changes to the structure. This requires re-parsing following all user edits – something which could seriously interfere with editing unless it can be done very fast. We solve this problem by dividing parsing into two phases.

In the first phase, a fast lexer is used to perform syntax highlighting and to break scripts into smaller partitions as the user is typing. This uses a simple scanning algorithm, and is based on Eclipse's standard classes.

In the second phase, the PGIP mark-up structure is obtained using a "deep" parser. Typically, the Deep Parser module will act as a go-between, and the actual parsing will be done by the prover or another outside process (c.f. Fig. 7) – although our implementation also allows for a Java-based parser to be used, if one is available (which could be more efficient). The deep parser is not fast enough to be run as the user types, but it can be run in the background in a low priority thread, or as needed for specific user commands (such as evaluating a script).

4.4 Converting Isabelle to use PGIP

Adapting a prover to use PGIP (and hence the PG/ECLIPSE interface) can be done either within the prover or via a PGIP wrapper. The main work (besides XML wrapping and unwrapping, which can be done using standard libraries) is building a parser for the prover's scripting language. For many proof languages, this should be a straightforward task. Converting Isabelle presented more of a challenge due to the complex and extensible nature of it's syntax.

It was decided to build the parser component within Isabelle. While straightforward in principle, this turned out to be harder than expected. Isabelle/Isar interprets scripts using a mix of abstract parsing and operational semantics, and there were difficulties with parsing proof scripts independently of their execution: the Isabelle code uses functional combinators to build combined parse-execute functions that were hard to unravel.

We expect that this will usually be easier to do in other systems (and PG/KIT also supports standalone parsing components).

4.5 Displaying prover output

There are two facilities available for attractively displaying output from the prover. Firstly, the output uses the same symbol support as the text-editor,

allowing it to contain mathematical symbols. Secondly, the output is displayed using an embedded html display widget, and can be formatted in arbitrary ways via an XSL style-sheet. The default style-sheet provides pretty-formatting (e.g. colouring of free vs. bound variables) for the PG *Markup Language* (PGML). This is a sub-protocol of PGIP, which can be used to mark-up display messages. PGML combines lightweight mark-up for terms with meta-information on how terms should be displayed.

5 Related work

A notable exception to the rule that theorem provers have poor front-ends is `Theorema`, a theorem prover built on top of `Mathematica` [**?**]. This provides it with an excellent user-interface for entering and viewing mathematical expressions. However this is also a weakness, as Mathematica is proprietary software with closely-guarded source code. This means its GUI cannot really be extended, and the `Theorema` system cannot be made freely available.

Various systems already exists for script management (e.g. PG/Emacs, or the `PCoq` interface for `Coq` [1]). However, these offer less functionality, and moreover do not provide a solution to the problem of handling different provers.

The MathWeb project provides a standardised XML-RPC interface to a range of automated provers (Otter, Spas, etc.) [**?**]. However it does not yet have support for interactive systems (partly because it has no inherent concept of a proof state or a proof session), or provide more than a very basic interface. This makes it a valuable possible partner to Proof General, rather than an alternative. MathWeb uses the XML format OMDoc as an exchange language [10]. OMDoc explains the semantical content of logical terms, which goes beyond the PG/Kit. It would be intriguing to consider an extension of our protocols to allow OMDoc exchange, although we would not want to force the underlying provers to implement OMDoc support.

In addition to MathWeb, there are several other efforts to publish formalised mathematical content, including Mizar [17], HELM [2], MoWGLI [12] and Logosphere [14].

The most closely related work is that done within the PG/Kit framework. The 2nd author has developed a PGIP-based version of PG/Emacs, whilst the 3rd author is working on a PGIP-based 'proof desktop', where theories and proofs are built up using graphical actions such as drag-and-drop. There is also a *broker* component in that can act as a middle-man between a collection of PGIP-equipped provers and interfaces [**?**]. This should lead to increasingly flexible ways to develop proofs.

6 Future work

There are many possible lines for future development.

Firstly, we want to use the Eclipse framework to further explore the analogy between theory development and software engineering. Ideas such as *code*

folding (which makes large files more navigable by allowing blocks of a proof script to be hidden (folded away)), *refactoring* (i.e. support for renaming and reorganising sections of code), type checking, etc. could usefully be applied to proof development tools.

PG/KIT *development support* The full PGIP spec is quite large. We are looking at specifying smaller and simpler subsets of PGIP for each component. This should make developing for PG/ECLIPSE more straightforward. We will also look at providing support for moving existing provers to the PG/KIT.

6.1 Proof Planning support

We can also go beyond adapting program development tools. One promising line of work is on using interactive proof planning to construct proof scripts [6].

Proof planning is a powerful backwards-reasoning technique based on capturing expert knowledge of proof structures. Although currently only used in the automated proving community, it could be a valuable idea to transfer to IDE based programming. The closest analogue is the idea of 'programming templates'; fragments of code (e.g. a `for` loop, or a standard widget initialisation method) which can be used to help build up programs.

However proof planning is considerably more powerful. Whilst programming templates typically provide only a starting point for coding, proof plans are generally more recursive, and the technique can be useful at any level of the proof. Moreover, proof-planning systems can include automatic search (often utilising heuristics such as *rippling* [**?**]), and automatic analysis of failed proof attempts [**?**]. We are currently working with Lucas Dixon on integrating IsaPlanner (an Isabelle based proof-planner) with PG/ECLIPSE [**?**]. This should lead to a PGIP-based API for generic proof-planner support.

The proposed UI for this is:

1. The user asks for help on how to tackle an unsolved goal. This can be the current goal, but it could also be any goal in the script which is marked as unsolved. For example, in the script shown in Fig ??, the user could select the 'sorry' (which marks an unsolved goal in Isabelle) via a right-click activated context menu. In this case, PG's 'goto' command is used to process or undo the script as necessary.
2. The planner responds with a list of suggestions, comprising possible changes to the script. At the simplest level this could be a list of steps to try, but it could also include advice on lemma-speculation, or generalisations that might be easier to prove ([**?**]).
3. The user can then select one of these suggestions, or ignore them. If a suggestion is selected, then the proof script text is edited accordingly.

Often a proof plan will include unsolved sub-goals, as happens in the example shown in 6.1. In this case, the process can be repeated if desired. This modus operandi allows the user to switch seamlessly back and forth between free-form text-editing and using a proof planner.

Fig. 8. A partial Isabelle proof is extended using a proof-planner.

In related work, Alex Heneveld is developing a planning view extension for PG/ECLIPSE that will use a tree to show the current exploration of the search space.

6.2 Longer term aims

The PG/KIT framework should provide a good platform for developing a variety of ideas to enhance and improve interactive theorem provers. Ideas we are considering include:

- Greater support for mathematical layouts (e.g. fractions).
- Diagrams provide intuitive ways of representing and reasoning about many domains. They are ubiquitous in mathematics texts, but hardly used in theorem provers.[9] Those provers that do use diagrams (e.g. HyperProof [**?**] or Dr.Doodle [7]), are currently very domain specific, and not suitable for serious mathematical research. It would be a major breakthrough to develop support for diagrammatic editors and views that would allow them to be used as part of proof texts.
- Supporting formal proofs that utilise multiple provers. This requires translation mechanisms, which are inherently logic-specific (or worse, specific to a 2-prover combination). It also requires great care if the guarantee of soundness is to be preserved. Nevertheless, there are some standards which might make this feasible (e.g. `Otter` syntax is understood by many 1st order theorem provers). This goal would probably be realised by a link-up with the MathWeb project.
- User-friendly enhancements to proof languages. The Proof General framework already partially separates the proof language seen and edited by the user (the 'user level language') from that seen by the broker/prover (which uses PGIP). Thus this framework is well suited to developing extra language layers or allowing user level proof languages to be modified without having to re-program the underlying theorem prover.
  This could be used to support productivity-enhancing features such as latex style macros. Macros would be easy to develop within the PG/ECLIPSE framework. However, since there is not a well-defined usage for macros in proof scripts, we do not want to commit to a macro-language without further investigation.
  More ambitiously, it would be interesting to investigate natural language rendering and interaction at a generic level [**?**]??.
- Another direction would be to develop a document-driven methodology, where users write papers that contain verifiable proofs [16].

[9] Apart from *proof trees*, which – whilst valuable as an interactive representation – are notable for their complete absence from textbooks.

7 Conclusion

PG/ECLIPSE provides ??

We have outlined the main features of the system, and given details on how these were implemented.

An important aspect of the implementation is that it is based on the PG/KIT framework. This framework is designed to aid the development of both provers and proof-interfaces. This framework provides a clearly defined API for connecting interactive proof systems to interface tools. Ultimately, we hope that implementers of both new and existing proof systems will have a compelling reason to add PGIP support to their systems to access powerful front-ends. On the other end, we hope that the PG/KIT will stimulate development of a wider range of interface tools. Such tools could be developed as Eclipse plugins – extending PG/ECLIPSE– or as stand-alone components.

7.1 State of the project

An alpha-release of PG/ECLIPSE is now available. It can be downloaded from http://proofgeneral.inf.ed.ac.uk/kit/wiki. We plan to issue a beta-release in January 2005. The latest version of Isabelle now supports PGIP (due to work by the 2nd author), and is available from [?]. Developers interested in using Proof General for their own provers should contact the authors. We also welcome contact from researchers interested in working with us on future directions.

Acknowledgments: D.Winterstein was supported by a 2004 Eclipse Innovation Grant awarded by IBM. D.Aspinall benefited from support provided by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing. C.Lüth does it for love.

References

1. A. Amerkad, Y. Bertot, L. Rideau, and L. Pottier. Mathematics and proof presentation in Pcoq. In *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, 2001.
2. A. Asperti, L. Padovani, C. S. Coen, and I. Schena. HELM and the semantic math-web. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2001.
3. D. Aspinall. Proof General: A generic tool for proof development. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1785, pages 38–42. Springer, 2000.
4. D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General system documentation, 1999-2004. Available at http://proofgeneral.inf.ed.ac.uk/doc.
5. D. Aspinall and C. Lüth. Commentary on PGIP. Available from http://proofgeneral.inf.ed.ac.uk/kit/, September 2003.

6. L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics 2004 (TPHOLs'04)*, volume 3223 of *Lecture Notes in Computer Science*. Springer, 2004.
7. D.Winterstein. Dr.doodle something or other?? 2004.
8. Eclipse homepage: eclipse.org. See http://www.eclipse.org.
9. H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In *TYPES*, pages 96–111, 2000.
10. M. Kohlhase. OMdoc: Towards an OpenMath representation of mathematical documents. Available from http://www.mathweb.org/omdoc/.
11. D. Kroening. *Formal Verification of Pipelined Microprocessors*, pages 71–80. Gesellschaft fuer Informatik, 2002.
12. MoWGLI. mathematics on the web: Get it right by logics and interfaces. http://www.mowgli.cs.unibo.it/.
13. RELAX NG xml schema language, 2003. Home page at http://www.relaxng.org/.
14. C. Schürmann, F. Pfenning, M. Kohlhase, N. Shankar, and S. Owre. Logosphere. a formal digital library. http://www.logosphere.org/, 2003.
15. S. Shavor, J. D'Anjou, S. Fairborther, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003.
16. L. Théry. Colouring proofs: a lightweight approach to adding formal structure to proofs. In D. Aspinall and C. Lüth, editors, *User Interfaces for Theorem Provers UITP'03*, volume 103 of *Electronic Notes in Theoretical Computer Science*, 2003.
17. A. Trybulec et al. The mizar project, 1973. See web page hosted at http://mizar.org, University of Bialystok, Poland.
18. D. von Oheimb and T. Nipkow. Machine-checking the java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, pages 119–156, 1999.
19. C. Wedler. Emacs package X-Symbol. Available from http://x-symbol.sourceforge.net, 2003.