Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit

Makarius Wenzel[1,2]

Université Paris-Sud 11, LRI
Orsay, France

Abstract

After several decades, most proof assistants are still centered around TTY-based interaction in a tight read-eval-print loop. Even well-known Emacs modes for such provers follow this synchronous model based on single commands with immediate response, meaning that the editor waits for the prover after each command. There have been some attempts to re-implement prover interfaces in big IDE frameworks, while keeping the old interaction model. Can we do better than that?
Ten years ago, the Isabelle/Isar proof language already emphasized the idea of *proof document* (structured text) instead of *proof script* (sequence of commands), although the implementation was still emulating TTY interaction in order to be able to work with the then emerging Proof General interface. After some recent reworking of Isabelle internals, to support parallel processing of theories and proofs, the original idea of structured document processing has surfaced again.
Isabelle versions from 2009 or later already provide some support for interactive proof documents with asynchronous checking, which awaits to be connected to a suitable editor framework or full-scale IDE. The remaining problem is how to do that systematically, without having to specify and implement complex protocols for prover interaction.
This is the point where we introduce the new Isabelle/Scala layer, which is meant to expose certain aspects of Isabelle/ML to the outside world. The Scala language (by Martin Odersky) is sufficiently close to ML in order to model well-known prover concepts conveniently, but Scala also runs on the JVM and can access existing Java libraries directly. By building more and more external system wrapping for Isabelle in Scala, we eventually reach the point where we can integrate the prover seamlessly into existing IDEs (say Netbeans).
To avoid getting side-tracked by IDE platform complexity, our current experiments are focused on jEdit, which is a powerful editor framework written in Java that can be easily extended by plugin modules. Our plugins are written again in Scala for our convenience, and to leverage the Scala *actor library* for parallel and interactive programming. Thanks to the Isabelle/Scala layer, the Isabelle/jEdit implementation is very small and simple.

Keywords: Isabelle, Scala, jEdit, asynchronous proof processing, re-use of editor and IDE frameworks

[1] Research partially supported by BMBF project "Verisoft" (01 IS C38)
[2] makarius@sketis.net

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

1 Introduction

From TTY loop to Proof General

Interactive provers in the tradition of Milner's LCF are still notorious for the lack of sophisticated user-interfaces. The original LCF system featured a TTY-based read-eval-print toplevel for the ML programming language, and later systems, such as Coq [15, §4] and Isabelle [15, §6] have reformed this only mildly, e.g. by introducing their own command language beyond ML.

If we reconsider the well-known Proof General / Emacs interface [2], which can be counted as a big success for a variety of provers, its interaction model is closely tied to the TTY model. Prover commands are issued one after another in a sequential (synchronous) manner, while each result is displayed immediately in separate windows (typically one for goal output and one for other responses). The main editor window is split into two parts: a *locked region* of text that has been processed already, and an editable region of unprocessed text. Thus Proof General provides a simple metaphor of stepwise execution of a *proof script*. The user can move the frontier between the two editor regions via *assert* and *retract* commands.

This model fits nicely onto the toplevel loop of many existing provers. The main additional requirement is an *undo* facility for backward movement. Note that the original HOL family [15, §1] lacks that, so Proof General support has always been very limited there.

Interaction in the style of Proof General greatly enhances the TTY model, but is still centered around the idea of *one command after another*. For example, the SSREFLECT scripting language for Coq [15, §4.2] capitalizes on quick typing of many small commands, with immediate feedback from the prover via proof state output. Even with Proof General as the preferred interface, this is TTY-style interaction *par excellance*.

Beyond Proof General?

Proof General has been able to dominate the interactive theorem community for many years. Its basic interaction model has been imitated several times, e.g. by CoqIde [15, §4.6], ProofWeb [10], or Matita [1]. There are further clones of Proof General that are not widely used.

Does this mean the Proof General model and its typical implementations are the final word on user-interfaces for theorem provers? We see two main movements to challenge its predominance in recent years, although without implementations that are ready for end-users so far. We observe the following main weaknesses of Proof General.

(i) Weaknesses of the underlying editor framework.
Classic Proof General uses the relatively powerful Emacs environment, although that has grown quite old. Some branches like XEmacs are

practically unmaintained. The GNU Emacs 23 branch has caught up in the past few years, but it still appears somewhat archaic to current user generations. There are also fundamental limitations of the Emacs platform, such as the single-threaded execution model of its LISP engine.

Most Proof General remakes use an even weaker editor: CoqIde implements its own editor in OCaml using existing GTk widgets. This is adequate for beginners, but many Coq power users switch back to Proof General / Emacs at some point. Although OCaml/GTk is multi-threaded in principle, CoqIde implementers have reported stability problems with the integration of GUI components and user threads. Moreover, OCaml does not support truely parallel execution of threads on multicore hardware, limiting the application to a small fraction of the CPU resources.

There have been various attempts to transplant the main ideas behind Proof General to full-scale IDE environments, notably Proof General / Eclipse (mostly for Isabelle) [4], Provereditor (for Eclipse and mostly Coq) [5], and I3P [8] (for Netbeans and mostly Isabelle). The hope is to replace relatively simple editor facilities by fully-featured IDE support, although this poses extra challenges due to the sheer size of typical IDEs.

(ii) Weaknesses of the interaction model.

As explained above, we can understand Proof General interaction as an add-on to plain TTY mode. In particular, there is only a *single focus*, which is the point where the last command has been successfully executed and the system awaits the user to enter the next one. This corresponds to the *prompt* of the TTY loop, and marks an inherently sequential / synchronous protocol.

Much less attention has been payed to this conceptual limitation so far, which we would consider more important than questions about the underlying editor platform. In Proof General / Eclipse [4] the sophisticated protocol definition of PGIP merely codifies classic Proof General interaction. Beyond that, some more recent work [3] explores *multi focus* editing in PLATΩ, based on concurrent XML update protocols on document sub-trees.

This situation is our starting point for the Isabelle/Scala and Isabelle/jEdit project. We would like overcome the limitations of traditional Proof General, both wrt. the underlying editor technology and the interaction model.

In §2 we outline an asynchronous interaction model that continuous our previous work on parallel proof checking in batch mode. In §3 we introduce the Isabelle/Scala layer as a mediating library between the ML and JVM world that provides important abstractions from raw inter-process communication. In §4 we report on our example application Isabelle/jEdit that draws on the results of §2 and §3. Further related work is referenced as we go along, while covering particular aspects.

2 Parallel proof checking and asynchronous interaction

Multi-threading for sophisticated GUI applications is important for two reasons. First, it helps to structure an interactive application systematically, e.g. to avoid blocking user input. Second, the advent of mainstream multi-core hardware about 5 years ago has made parallelism inevitable for any CPU intensive application.

The problem of efficient parallel proof checking in batch-mode has been addressed already for Isabelle and the underlying Poly/ML platform, see [13] and [11]. The following main virtues of our prover and the ML implementation have allowed to retrofit parallel proof processing onto the existing system.

- *DAG-structured development graph of the theories that form a project.* This enables simple concurrent loading in the style of parallel `make` tools, such as GNU `make -j`. Little needs to be changed in the system to achieve such parallelism at the outermost level, but the speedup factor is limited by theory dependencies, which are often relatively linear.
- *Fully-specified toplevel theorems and proof irrelevance for most practical purposes.* The original LCF family has been built around the idea of an inference kernel that merely checks proofs without necessarily keeping an explicit record of them. Even in type-theory based systems like Coq, proofs are usually *opaque* and not taken into account in further proofs that refer to the resulting theorems. Whenever resulting propositions are specified in advance, we can easily fork the proof process, and join everything in the very end of loading a whole sub-graph of theories. This scheme allows to saturate ≈ 4 cores quite well [13, §5].
- *Strictly modular proofs in the structured proof language of Isar.* This is a special bonus for well-structured Isar proof texts. The design of the Isar proof language follows the principle that the main outline can be checked quickly, while time consuming automated reasoning is limited to justifications of terminal sub-proofs, e.g. "**by** *simp*" or "**by** *blast*". The latter can be forked / joined in a similar manner as above.

  This extra potential of parallelism turns out practically important beyond 4–8 cores [11, §5]. In large applications like *JinjaThreads* in AFP `http://afp.sf.net/entries/JinjaThreads.shtml`, parallel checking of sub-structures makes a big difference, with total runtime of 30 min for toplevel parallelism vs. 8 min for parallelism also in sub-proofs.
- *Isabelle/ML programming style based on immutable values.* Thanks to mostly clean implementation of the main parts of Isabelle, with immutable data structures almost everywhere, it has been relatively easy to change the underlying execution model at a grand scale. Only small portions of impure code had to be eliminated.

To summarize, parallel batch processing of Isabelle theories and proofs is relatively easy to achieve in principle, since it can be understood as a suitable reorganization of the "evaluation process" of certain symbolic proof expressions. The hard part is to build basic infrastructure for parallel ML from scratch [13][11], and to get a reasonable speedup factor in the end (such as 3.0 for 4 cores, and 5.0 for 8 cores).

How does parallel proof processing affect the interaction model? We have already explained why the TTY model is inherently sequential, working slowly on one command after another and waiting for results from the prover until the current command is finished, as indicated by the toplevel command prompt.

In order to exploit the rich theory and proof structure not just for parallel batch processing, but also in interactive mode, we need to rethink the prover toplevel itself. In Isabelle2009/2009-1, there is already a simple asynchronous toplevel that supports an *asynchronous document model* natively as follows.

- Explicit operations *begin_document* and *end_document* delimit the scope for any further operations below. Such a document is understood as a persistent entity, with purely functional update operations. There are explicit version identifiers. Operating on the initial root created via *begin_document* produces a tree of versions evolving over time. A final *end_document* chooses a single *success path* to be committed to the theory database eventually.
- The *define_command* operation identifies a piece of source text ("command span" in Proof General terminology) for later use. Such a command essentially represents a function on the semantic state of the prover (which includes the theory and proof context, goal state etc.). Interaction with the prover means to compose such functions in various ways, each resulting in a partially evaluated proof attempt.
- The *edit_document* operation updates a document via *insert* and *remove* primitives of commands as defined above. This results in a new document with a fresh identifier, and the prover will report internal state changes eventually. The asynchronous toplevel maintains a multitude of such related document versions, which typically share common sub-structures. Results emerging from a new document version are reported as they arrive, according to the parallel evaluation process managed by the prover. Messages are explicitly identified by the corresponding command within a certain document version (the toplevel prompt is abolished).

The above primitives roughly sketch a protocol between the editor and the prover that can give the user the impression of continuous checking of text, providing prover feedback as it emerges incrementally. The editor never waits for the prover, and never stops the user from typing. The prover is free to schedule the evaluation of partial proof documents to exploit the potential of

parallelism as much as possible. Even without parallel checking, the prover can arrange document processing in a way that allows the user to edit proofs independently from each other, without costly replay of the whole script.

We expect great impact on the efficiency of interactive proof development by this decoupling of the mechanics of proof document processing. Nonetheless, it is still quite hard to connect an editor to the prover at this raw interface, so we introduce a more abstract programming API in the Isabelle/Scala layer.

3 The Isabelle/Scala library for prover interaction

In order to understand the key role of the Isabelle/Scala layer, we first reconsider the basic problem of linking two rather dissimilar worlds: the prover implemented in Standard ML (notably Poly/ML), using *pure* and *higher-order* functional techniques pervasively, and the somewhat more "industrial" Java runtime environment, with its mutable objects, null pointers, and awkward verbosity of the Java source language.

Note that Coq and Matita are implemented in OCaml instead, and there are basic GUI libraries for that platform that can be used as a starting point for some editor functionality. From what people behind CoqIde and the Matita GUI have reported privately or publicly [1], we conclude that OCaml/GTk is a minority GUI platform nonetheless, and there are technical issues that demand considerable development resources. Even if the basic GUI layer would work perfectly, advanced editors or IDE frameworks are still lacking.

Our genuine task is to build provers, not to re-implement editors or IDEs from scratch. So we prefer to link to existing frameworks, even though this means to accommodate the split into two different processes: ML vs. JVM. Explicit inter-process communication definitely raises some additional questions, and the inhomogeneous language situation complicates things further.

PGIP [4] addresses these issues by defining explicit protocols (using XML syntax) between various autonomous components. These components coincide with the separate programming language environments involved here: the *prover* process in ML, the *editor* in Java, a *broker* in Haskell. [3]

We argue that cutting the conceptual components at these process boundaries complicates the system integration, even without the (optional) broker in between. The protocol suite defined here covers many accidental details that the prover or editor happen to expose at their respective process boundaries. Implementing and maintaining such complex protocols is very hard.

Our approach to bridge the gap between the prover and the editor is quite

[3] Interestingly, the PGIP broker is motivated at some point as a means to enhance certain prover functionality, without having to modify the prover itself, and without having to struggle with sophisticated symbolic computations in Java.

different, see Figure 1. Instead of complex protocols, we postulate a relatively simple API on each side, to connect both worlds via a conceptual interactive document model in between. Operating on the document works via statically-typed library functions instead of raw protocol messages.

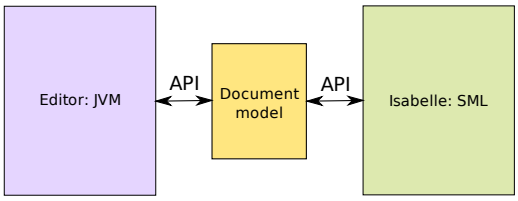

Fig. 1. The mediating document model between editor and prover — conceptual view.

Figure 2 explains how this can be actually implemented. There is an *internal protocol* that is private to the library implementation; only considerable abstracted concepts are exposed in the APIs on either side. The split of the library implementation into two parts (ML vs. JVM) is addressed by maintaining these dual-language modules side-by-side, within the main source tree of Isabelle/Pure, and with identical names for the corresponding ML structures/functors vs. Scala objects/classes/traits. The flexibility of Scala [12] allows to imitate the Isabelle/ML programming style on the JVM side. [4]

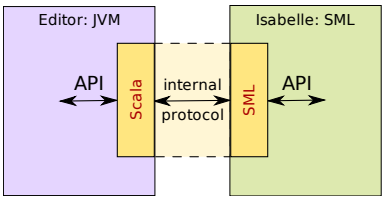

Fig. 2. Document model implementation via internal protocol and adapter library.

For example, to represent abstract XML trees, `xml.ML` defines a datatype in 3 lines of code, and various supporting functions in a few pages. The corresponding `xml.scala` uses 3 lines of Scala *case classes*, and various basic methods (on immutable objects) of similar size than the ML version. Naturally, there is a considerable overlap in functionality, but each side also has its distinctive parts. For example, the Scala side provides a function to turn pure XML trees into an official (mutable) `org.w3c.dom.Document`, because some existing Java components happen to require this occasionally, say an HTML rendering engine.

To transfer these ubiquitous XML trees between the two processes, the internal protocol uses the simple and efficient YXML encoding [14, §4.12]. Thus we scale-down the daunting task of fully official XML document exchange

[4] Higher-order functional programming in Scala works very well, with a reasonable code inflation factor of 1.5–2.0 compared to best-practice Isabelle/ML.

to a very simple transfer format of our own invention, which can be specified on 1/2 page and implemented efficiently in 1/2 day.

Keeping the often delicate details of inter-process communication private has further practical advantages. In particular, we can change the protocol easily in order to adapt it to new requirements. In the brief history of the Isabelle/Scala layer, the internal protocol was subject to several substantial changes already, both concerning central ideas of the underlying asynchronous document model, and marginal details for improved performance and robustness. Such protocol refinements merely require a single party to agree with itself, instead of a protocol committee negotiating over and over again.

Potential incompatibilities at the Isabelle/Scala API level can be accommodated in client code easily, because the signatures are statically typed.

Beyond mere connectivity

There is more to Isabelle/Scala than simple and robust connectivity of the ML and JVM worlds. When working on the Isabelle/jEdit application, the immediate GUI programming requirements on the editor side turned out to be marginal compared the general notion of persistent documents that can be updated asynchronously. This means the conceptually deeper parts are independent from the particular editor and can be addressed generically within the Isabelle/Scala library. The basic idea is to model families of immutable documents under version control, using ML-like Scala structures (tuples, lists, options, functions). Concurrency and true parallelism is achieved via the *actor library* of Scala [9], which provides a very nice abstraction of independent functional entities linked by explicit message channels. Following the principle of side-by-side modules of ML vs. Scala again, we also provide some simple add-ons to the actor library, to imitate the value-oriented parallelism around *futures* in Isabelle/ML [11, §4].

These more advanced aspects of the asynchronous document model will require further elaboration in future work. For example, the present history only works for a single theory, and ignores the implicit theory graph that is required for multi-buffer editing. Nonetheless, the present Isabelle/Scala layer is sufficient to support our Isabelle/jEdit application, and enables some practical assessments of the general approach.

4 Application: Isabelle/jEdit

Before we discuss the internal structure of Isabelle/jEdit we first illustrate how it presents itself to the end-user, see Figure 3. It is hard to capture the dynamic process of asynchronous proof editing in static screenshots. The general look-and-feel is similar to existing Java IDEs in Eclipse or Netbeans:

the user types text as he pleases, and the editor provides useful feedback incrementally, via semantic information from the partially processed document in the background. This achieves *continuous proof checking* based on our asynchronous prover toplevel. A lot of information can be directly attached to the source text, via coloring, tooltips, popups etc. Thus we address the challenge of proof editing without the goal state buffer posed in [6].

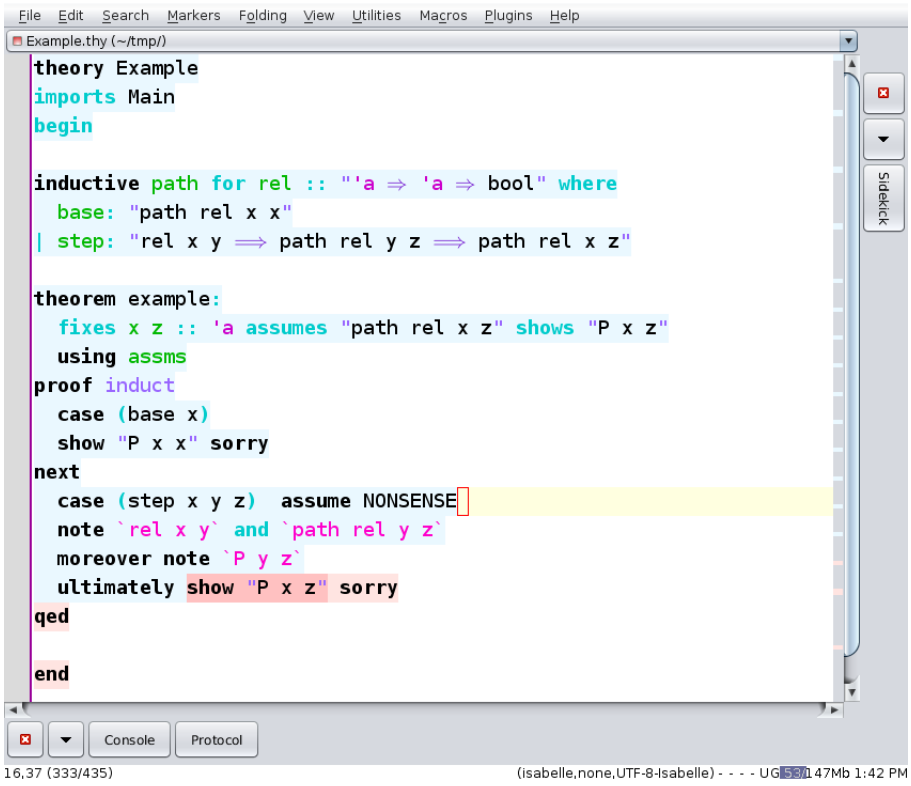

Fig. 3. Main editor window of Isabelle/jEdit, with semantic highlighting

We also provide traditional prover output windows apart from the main editor frame, e.g. see Figure 4 for a proof state display that corresponds to the last **show** command (in red).

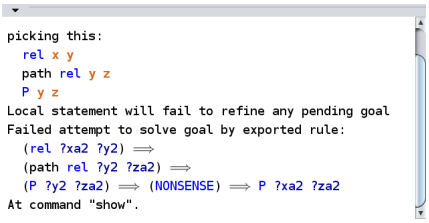

Fig. 4. An instance of the Output dockable showing a proof state

Actually, the error reported in the result message of **show** is caused in a different spot of the text, namely the nonsensical **assume** statement at the

cursor position, which is not admissible in that proof context. The dynamic change of colors already indicates such a structural error in the text: after inserting that **assume** command, the corresponding **show** command turns red. By highlighting such non-local errors directly in the text, we have already transcended the Proof General model at a small scale. Further refinements will be required to achieve a viable editing model for Isar proofs in particular — Isar had been locked into the TTY loop for more than 10 years already.

4.1 The jEdit editor framework

The *jEdit* editor `http://www.jedit.org` is advertised as "a mature programmer's text editor with hundreds of person-years of development behind it". Compared to fully-featured IDEs, such as Eclipse or Netbeans, jEdit is much smaller and better focused on its primary task of text editing. This main jEdit functionality is provided by `jedit.jar` (4 MB size in jEdit 4.3.2), and numerous plugins can be downloaded from a repository. As is typical for such frameworks the quality of plugins varies greatly, ranging from major add-ons written by core jEdit developers to adhoc experiments by interested users.

Plugins

A jEdit plugin consists of the main JVM object-code as a regular *jar*, together with some meta-data via JVM property files and some judicious use of XML. Components are glued together via code snippets in BeanShell, which is essentially a light-weight interpreted version of Java. There is even an interactive BeanShell console (as a standard plugin) that exposes the name space of JVM classes of the running editor environment. This is reminiscent of the Emacs scratch buffer, with its direct access to the LISP runtime environment.

Getting started in implementing jEdit plugins is very easy by studying documentation, example plugins, and the sources of the editor and some major plugins. The jEdit console plugin greatly helps interactive exploration and debugging. Alternatively, the whole application can be run under control of existing JVM debugging tools, say the Java source-level debugger of Netbeans.

Integrated applications

Apart from using the official jEdit distribution together with some plugins, it is also possible use jEdit as a basis for an integrated application that is targeted for specific purposes. For example, MathPiper `http://www.mathpiper.org` provides a "Mathematical IDE" by bundling jEdit with some other mathematical applications, including the well-known GeoGebra system `http://www.geogebra.org` for dynamic geometry, and some components for computer algebra and chart drawing. In this scenario MathPiper starts a cus-

tomized version of jEdit with some preloaded plugins run in sub-windows of the "dockable window manager" of jEdit.

A similar derivative application is OQMathJEdit from the ActiveMath project http://www.activemath.org. The CZT suite http://czt.sourceforge.net also includes some jEdit plugins that can be used as the basis for a small-scale IDE for the Z specification language.

Compared to large-scale IDE frameworks, such tool integration is very simple, as long as the components are available as regular *jars*. Historically, this usually meant Java as implementation language, but recently some alternative JVM-based languages have attained sufficient attention to be counted as "high-profile languages" by the JVM community. Apart from various scripting languages such as Groovy, Jython, JRuby, two functional languages have gained attention in the mainstream world: Clojure http://clojure.org (roughly an untyped version of Haskell using LISP notation), and Scala http://www.scala-lang.org with its sophisticated integration of higher-order functional object-oriented programming and direct access to existing Java class libraries.

4.2 Isabelle/jEdit

The main Isabelle/jEdit plugin consists of ≈ 10 small Scala files (42 KB total size) that augment some key jEdit components in order to provide a metaphor of asynchronous proof document editing as outlined before.

Isabelle/jEdit integrates the jEdit 4.3.2 framework with our own Isabelle plugin (written in Scala), and some further off-the-shelf jEdit plugins. The main jEdit distribution is essentially unchanged, but the whole is presented to the user as a "pre-canned" jEdit installation that can be started immediately via the regular Isabelle tool wrapper. Apart from certain default properties, the startup phase of the Java runtime and the core jEdit component is fine-tuned to take care of important details such as Isabelle-specific text encoding *UTF-8-Isabelle*, and a custom-made `IsabelleText` Unicode font that actually contains the usual Isabelle symbols that users expect from long years of Proof General X-Symbol support.

Isabelle plugin components

The core functionality is directly attached to the key editor concepts of jEdit as follows.

- Our `Document_Model` augments a jEdit `Buffer` by semantic information provided by the asynchronous document model of Isabelle/Scala in the background. We maintain a partial function from `Buffer` to `Document_Model` — every buffer that is associated with an Isabelle theory file is "enhanced"

by adding an instance of `Document_Model`. [5]

Since the actual document model is maintained by the Isabelle/Scala layer, the editor side needs to do very little here. The main responsibilities of the `Document_Model` are as follows.

· Maintaining a pointer into the persistent history of the Isabelle/Scala proof document model. This affects both *input* and *presentation* of text: all operations are relative to this explicitly identified point in history, which is represented as a unique version identifier.

· Input wiring involves a regular *event handler* for *insert* and *remove* actions on plain text produced by jEdit. These events are locally buffered, and eventually forwarded to the underlying document model together with the history identifier. The Isabelle/Scala library turns text edits into larger chunks based on the command structure of the proof language.

This multi-stage pipeline of edits decouples the editor from the prover: there is no locking nor "object ownership" [7] involved here, which would make the process more synchronous than necessary. The user can type into the editor at any time, independently of the prover's responses that might arrive only much later, say within 20 ms ... 200 ms.

· Presentation wiring involves a *token marker* associated with the buffer that implements semantic syntax highlighting, meaning that authentic information from the prover is used to indicate the formal status of certain pieces of text. This avoids the typical frustration of editor-based syntax tables that approximate syntax highlighting in a crude way. Here we use the information provided by the prover, the only instance that really understands its own syntax. Thus we can highlight the term language of Isabelle that is embedded as "inner syntax" in the theory source, or Isar text with embedded ML which in turn refers to formal entities via antiquotations, and all that interspersed with some nested comments.

- Our `Document_View` augments a jEdit `TextArea` in a similar way as above. It covers the immediate visual aspects of presenting a text buffer, allowing multiple views on the same content.

  The jEdit `TextArea` enables plugins to provide custom *painters* that get direct access to Java graphics context to modify the visual appearance in almost arbitrary ways. We use this facility to represent the command status (*unprocessed*: pink, *finished*: faint blue, *failed*: red), both as background of the source text (as in Proof General), and as small ticks in the right margin of the text view (as in common IDEs).

  There is additional wiring to follow the cursor movements within the `TextArea`: it influences other plugin components, notably prover output windows.

[5] Luckily, the jEdit developers do not insist on a purely static class hierarchy, but provide a suitable backdoor that is reminiscent of plain-old object property lists.

We provide a few independent "dockable windows" that are integrated into the window manager of jEdit. These are easily implemented as sub-classes of Java Swing frames and declared to jEdit via some basic XML configuration. The editor manages any number of instances of such dockables, either as freely floating windows, or docked at a margin of the main editor view. Isabelle/jEdit provides the following dockable classes.

- The `Output` dockable displays result messages (including proof state) of the command where the cursor is pointing, using the point of history of the underlying `Document_Model`.

  Isabelle messages contain rich information represented as markup-trees. This is rendered by mapping it to XHTML internally, and letting the Lobo browser `http://lobobrowser.org` display it by means of a given CSS. Thus we gain quite flexible output facilities almost for free. Proof General style syntax highlighting of free vs. bound variables, type variables etc. can re-use the existing CSS for Isabelle HTML presentation of theory sources. Further semantic information provided by the prover, such as references to formal entities (types, constants etc.), can be linked to internal Scala operations to implement hyperrefs, although this is not fully activated yet.
- The `Protocol` dockable displays the raw stream of prover input and output messages using XML notation. Note that the internal protocol uses the more efficient transfer syntax of YXML. Significant slowdown is caused by printing all protocol messages explicitly within a Swing text component, so this is really for debugging only.

Off-the-shelf plugins

The jEdit repository also provides quite useful "meta-plugins" that can be easily instantiated for our purpose, say a generic tree view on the document model underlying the theory text, or a text console that can be re-used as Scala read-eval-print loop. Paradoxically, such basic functionality often needs to be re-implemented from scratch in larger IDEs, due to their broader approach as a platform for "everything and nothing in particular" (Eclipse). The Isabelle/jEdit application re-uses the following jEdit plugins.

- `Console` with our `Scala` sub-plugin to provide a regular read-eval-print loop. This uses the existing class `Interpreter` provided by the Scala compiler suite from EPF Lausanne. It is important to note that the interpreter really runs within the same JVM environment as the application itself.

  It might be arguable if end-users really need access to an application at the programming language level, but the Scala console already proved an inevitable development tool that is culturally very close to the conventional toplevel loops of OCaml or Poly/ML. Moreover, it provides immediate "scriptability" of the application, although the integration with the rest of

the framework is not as sophisticated as for BeanShell, which is the standard jEdit scripting language.

- `SideKick` to provide a tree-view on the source buffer, but also for completion popups and tooltips.
- `Hyperlinks` for simple clickable references in the source buffer.

Setting up such plugins for our purposes is usually quite simple. It requires to read some example sources (in Java), and to implement our prover-specific functionality typically in 0.5–2 pages of Scala. There are some further generic jEdit plugins that can be used directly in our context without requiring any additional configuration. For example, the `Highlight` plugin provides impromptu syntax highlighting based on regular expressions given by the user.

5 Conclusion

Bridging the cultural gap between ML and Java

From our own encounter with the JVM world in the past 2 years, and from discussions with people behind other prover interface efforts, we can say that there is clearly a cultural gap between these different worlds. Interactive provers are typically written in Standard ML, OCaml, or Haskell, using deep programming techniques based on higher-order principles (recursive λ-calculus with Hindley-Milner typing, monads, and recent trends even heading towards dependently-typed programming languages). In contrast, industrial-strength Java frameworks are more broad than deep, with huge IDEs and heavy-duty tooling to (re)generated code by "refactoring" etc. Few people are attracted by both ways of working, and even fewer are proficient in both at the same time. The prover community has occasionally tried to import GUI technology into their little world, typically using GTk in OCaml or Haskell. Nonetheless, we would say that even the relatively big communities behind OCaml or Haskell have not yet delivered access to mainstream GUI frameworks at a grand scale, and sophisticated editor or IDE frameworks are missing altogether.

Our answer to this cultural problem: Keep ML as clean implementation language for the prover, use Scala on the JVM for GUI/IDE connectivity. Higher-order programming in Scala works very well, and we gain access to interesting frameworks that happen to be implemented in a slightly odd language (Java). Scala also has quite nice standard libraries to offer, including *actors* as efficient model for parallel and interactive components.

Towards routine use of prover IDE technologies

Once the basic technical problems of connecting to contemporary IDEs are overcome, we need to elaborate on the genuine requirements for proof editing as opposed to established programming language IDEs. Fully-formal check-

ing down to the logical foundations has slightly different characteristics than static analysis of Java, for example. Our asynchronous proof document model addresses this by modeling unfinished proof attempts as first-class document versions, and by allowing individual proof steps to produce results incrementally, or even diverge. Further questions will arise when these ideas are scaled up to the level of library maintenance, ideally in combination with external version control, say via Mercurial with its efficient persistent history model.

Viable prover IDE support will be a prerequisite to enter these and other areas of formal proof in practical use. The PIDE manifesto http://bitbucket.org/pide/pide/wiki/Home provides some further perspective.

References

[1] Asperti, A., C. Sacerdoti Coen, E. Tassi and S. Zacchiroli, *User interaction with the Matita proof assistant*, Journal of Automated Reasoning **39** (2007).

[2] Aspinall, D., *Proof General: A generic tool for proof development*, in: S. Graf and M. Schwartzbach, editors, *European Joint Conferences on Theory and Practice of Software (ETAPS)*, LNCS **1785** (2000).

[3] Aspinall, D., S. Autexier, C. Lüth and M. Wagner, *Towards merging Plato and PGIP*, in: S. Autexier and C. Benzmüller, editors, *User Interfaces for Theorem Provers (UITP 2008)*, ENTCS **226** (2009).

[4] Aspinall, D., C. Lüth and D. Winterstein, *A framework for interactive proof*, in: M. Kauers, M. Kerber, R. Miner and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants (CALCULEMUS and MKM 2007)*, LNAI **4573** (2007).

[5] Charles, J. and J. Kiniry, *A lightweight theorem prover interface for Eclipse*, in: S. Autexier and C. Benzmüller, editors, *User Interfaces for Theorem Provers (UITP 2008)*, ENTCS **226** (2009).

[6] Dixon, L. and J. D. Fleuriot, *A proof-centric approach to mathematical assistants*, Journal of Applied Logic: Special Issue on Mathematics Assistance Systems **4** (2006).

[7] Gast, H., *Managing proof documents for asynchronous processing*, in: S. Autexier and C. Benzmüller, editors, *User Interfaces for Theorem Provers (UITP 2008)*, ENTCS **226** (2009).

[8] Gast, H., *Towards a modular extensible Isabelle interface*, in: S. Berghofer and M. Wenzel, editors, *Theorem Proving in Higher-Order Logics (TPHOLs) — Emerging Trends*, TU München, Institut für Informatik, 2009.

[9] Haller, P. and M. Odersky, *Event-based programming without inversion of control*, in: *Joint Modular Languages Conference*, Springer LNCS, 2006.

[10] Kaliszyk, C., *Web interfaces for proof assistants*, in: S. Autexier and C. Benzmüller, editors, *User Interfaces for Theorem Provers (UITP 2006)*, ENTCS **174** (2007).

[11] Matthews, D. C. J. and M. Wenzel, *Efficient parallel programming in Poly/ML and Isabelle/ML*, in: *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP 2010), co-located with POPL* (2010).

[12] Odersky, M. et al., *An overview of the Scala programming language*, Technical Report IC/2004/64, EPF Lausanne (2004).

[13] Wenzel, M., *Parallel proof checking in Isabelle/Isar*, in: G. Dos Reis and L. Théry, editors, *ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS)* (2009).

[14] Wenzel, M. and S. Berghofer, "The Isabelle System Manual (for Isabelle2009-1)," http://isabelle.in.tum.de/doc/system.pdf.

[15] Wiedijk, F., editor, "The Seventeen Provers of the World," LNAI **3600**, Springer, 2006.