System description: Isabelle/jEdit in 2014

Makarius Wenzel *
Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France

This is an updated system description for Isabelle/jEdit, according to the state of the official release Isabelle2013-2 (December 2013) and the forthcoming release Isabelle2014 that is anticipated for Summer 2014. The following new PIDE concepts are explained: asynchronous print functions and document overlays, syntactic and semantic completion, editor navigation, management of auxiliary files within the document-model.

1 Introduction

Isabelle/jEdit is a Prover IDE (PIDE) that integrates *parallel proof checking* [5, 11] with *asynchronous user interaction* [7, 10, 12], based on a document-oriented approach of *continuous proof processing* [6, 8]. This enables the user to edit whole libraries of formalized mathematics directly in the editor, with real-time visualization of feedback produced by the prover. Today Isabelle/jEdit is the default user-interface for Isabelle, but this has required many years of developing the PIDE concepts and getting the underlying Isabelle/Scala infrastructure into a robust and scalable state. The ultimate goal is to load the whole *Archive of Formal Proofs* [4] into a single IDE session, but that is growing at a high rate,[1] and there are still theory name space problems preventing that.

Although Isabelle/jEdit is the most visible Prover IDE application, and sometimes people erroneously attach the label "jEdit" to anything coming after the TTY loop and Proof General [1] in Isabelle, the PIDE principles are meant to be more general and applicable to other front-ends.

Isabelle/jEdit is an example for a *rich-client application* that is run on the local machine, with non-trivial resource requirements: 2–4 CPU cores and 2–4 GB memory minimum. An interesting alternative is the *client-server application* Clide [2, 3], which combines Isabelle/Scala/PIDE with recent Web technology on the JVM, and supports collaborative interactive theorem proving in particular.

Here is a brief historical overview of Isabelle/jEdit so far:

- In 2005 all major CPU manufacturers started to ship multicore systems for the consumer market. Ever since the burden of explicit parallelism has been imposed on application developers, in order to keep up with the changed side-conditions of *Moore's Law*, and participate in continued performance improvements of computing hardware.
- In 2006–2008 Isabelle and its underlying Poly/ML compiler / runtime system have managed to follow the multicore trend. Isabelle2008 (June 2008) was the first official release to support parallel proof processing in *batch mode* by default. At the same time it became apparent that user-interfaces for parallel proof assistants require significant reworking of the interaction model.

*Research supported by Project Paral-ITP (ANR-11-INSE-001).

[1] At the time of writing (26-May-2014), AFP consists of 176 articles in 1899 source files, which comprise 46 MB total. Checking in batch mode takes approximately 10h CPU time and 1h elapsed time on a solid 8-core Intel Xeon workstation.

Submitted to:
UITP 2014

- In 2010 the new Isabelle/PIDE concepts, with the underlying Isabelle/Scala infrastructure, and Isabelle/jEdit as experimental application, were presented in public at UITP 2010 [7]. That was 2 years after the first concrete ideas for it had emerged, but still more than 1 year to go before the first "stable release" of Isabelle/jEdit with Isabelle2011-1 (October 2011).
- In 2012 that initial release of Isabelle/jEdit was presented as system description and tool demonstration at CICM [8] and some of its concepts were explained at the co-located UITP 2012 [10].
- The subsequent releases of Isabelle/jEdit in May 2012, February 2013, and November/December 2013 have consolidated the PIDE concepts and its implementation. So many new things were introduced each time, that users have occasionally complained about having to learn a new Prover IDE with each Isabelle release.

It is important to understand that parallelism is a feature of last resort in hardware design: it makes it more difficult for application software to use the available CPU cycles. On the other hand, an application that works with parallel execution routinely opens new possibilities in interaction design.

2 Asynchronous print functions in Isabelle2013-2

The current release of Isabelle2013-2 (December 2013)[2] is notable, because it introduces support for *asynchronous print functions* in the PIDE document model. This combines *user interaction* and *tool integration* as explained in [12], which will be presented at ITP 2014 a few days before UITP 2014.

The general approach is to continue the reforms of READ-EVAL-PRINT [10] as follows [12, §5]:

- Edits may add or remove PRINT operations, without disturbing the corresponding EVAL tasks. This principle of *monotonicity* avoids interruption of tasks that are still active in the document model, and allows to use long-running or potentially non-terminating tools as print functions. Typically these are automated provers (via Sledgehammer) or disprovers (Quickcheck, Nitpick).
- Activation or deactivation of PRINT tasks is subject to the *document perspective*. The whole theory library that is edited might be big, but only small parts are visible in the editor. PIDE document processing takes the open text windows as indication where to invest resources for continuous processing. Various declarative parameters control print functions that are implemented in user-space of Isabelle/ML: startup delay, time limit, task priority, persistence of results within the document model.
- Support for explicit *document overlays*, which are print functions with arguments provided by some GUI components. This recovers the appearance of direct access to command execution in the prover, despite the thick layers of asynchronous PIDE protocol between the stateless/timeless prover and the physical editor.

The subsequent screenshots figure 1 and figure 2, which are taken from the Isabelle/jEdit manual [9], illustrate the use of asynchronous print functions and document overlays in practice.

Figure 1 shows various *automatically tried tools* that operate on outermost goal statements (e.g. **lemma**, **theorem**), independently of the state of the current proof attempt. They work implicitly without arguments, but there are global options in *Plugin Options / Isabelle / General / Automatically tried tools*. Results are output as *information messages*, which are indicated in the text area by blue squiggles and a

[2] `http://isabelle.in.tum.de/website-Isabelle2013-2`

blue information sign in the gutter of the text window. The message content may be shown as for other prover output in a separate window. Some tools produce output with *sendback* markup, which means that clicking on certain parts of the text, inserts that into the source in the proper place.

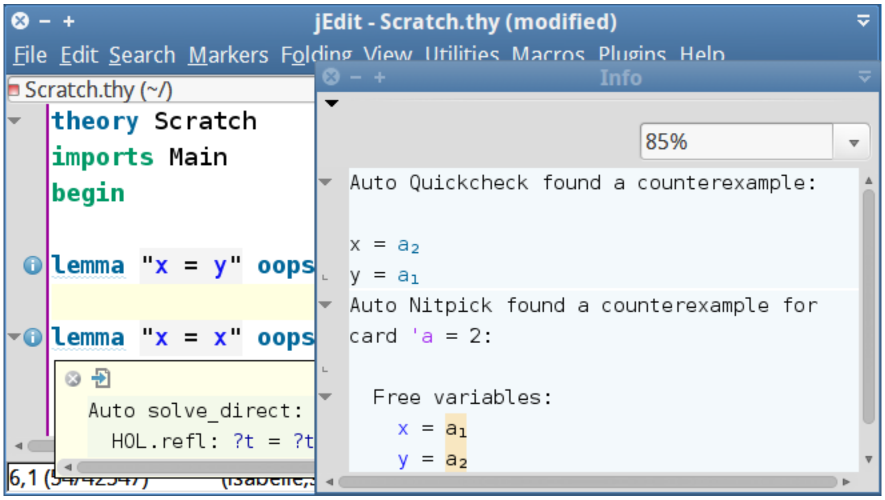

Figure 1: Results of automatically tried tools

Figure 2 shows the *Sledgehammer* panel, which provides a view on some independent execution of the Isar command **sledgehammer**, with process indicator (spinning wheel) and GUI elements for important Sledgehammer arguments and options. Any number of Sledgehammer panels may be active, according to the standard policies of jEdit window management. Closing such a dockable window also cancels the corresponding prover tasks.

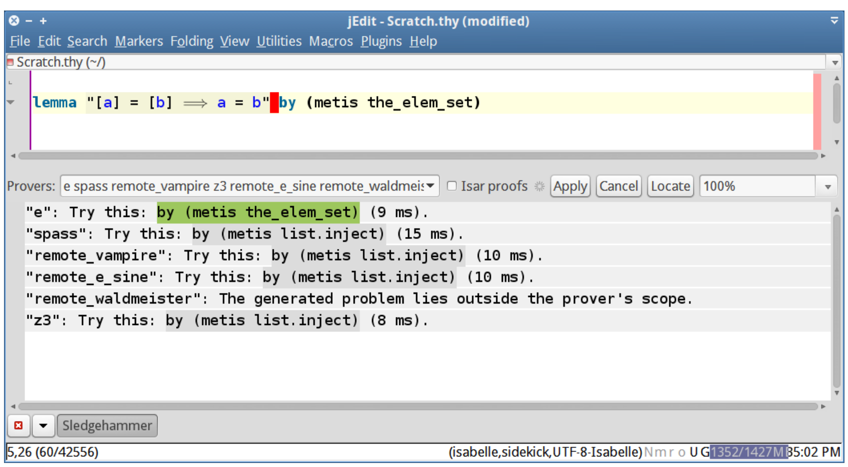

Figure 2: An instance of the Sledgehammer panel

Technically, the Sledgehammer panel is a conventional GUI component on the surface, but it is connected to the PIDE document model by producing some document overlay when the user pushes the *Apply* button. This leads to some document edit that attaches a suitable asynchronous print function (with arguments taken from the GUI panel), and forks some print task on the prover side. Any output from that task is incrementally shown in the GUI panel. The *Cancel* button uses the execution id of the running print operation to interrupt it on demand.

The overall interaction of the PIDE front-end with the prover back-end does not prevent the user from editing the text nor the prover from checking proofs in parallel. The only impact is some loss of performance to other tools in the background, but this can be balanced via global system options to adjust to the available number of cores.

3 Preview of Isabelle2014

The Isabelle2014 release is anticipated for Summer 2014[3]. There will be again an improved version of Isabelle/jEdit as the default Prover IDE, and some of its new PIDE concepts are described below.

3.1 Syntactic and semantic completion

Semantic completion, based on authentic information from the proof context, has been a "nice to have" features over several years. It was not immediately obvious to teach that trick to a traditional LCF-style proof assistant like Isabelle, which was not made for that 25 years ago.

Even just the editor GUI part of auto completion has turned out much less trivial than anticipated in 2009/2010 [7], where the (naive) idea was to connect to an existing completion plugin of jEdit. Over the last 5 years the completion mechanism in Isabelle/jEdit has changed several times, but various problems with the timing of GUI events still occur in Isabelle2013-2.

Completion intercepts the regular key event handling of the main text area, and needs to work smoothly as the user is typing slowly or quickly. The completion popup changes the keyboard focus to a different component, which can lead to odd effects of loosing key events in a situation where the user is typing fast, but the graphics display is too slow to catch-up (e.g. due to bad X11 rendering performance, which can happen both for local and remote displays).

The lesson learned here is that a Prover IDE is a highly interactive computer-game, with demands of real-time reactivity that were not present in TTY front-ends from the past.

Both the GUI event handling and the semantic aspects of completion in Isabelle/jEdit have been significantly reworked in the past few months. The following general principles are presently applied.

- **Syntactic completion** is based on information that is *immediately* available in the editor, e.g. keyword tables for certain sub-languages of Isabelle, like the so-called "outer syntax" of Isabelle/Isar, or Isabelle/ML. Completion for Isabelle symbols is an important a special case of this: when the user types "==>" he normally expects to get "$\Longrightarrow$" within formal text.
- **Semantic completion** is produced by the prover *eventually*, after a full round-trip through the asynchronous PIDE protocol. This information usually arrives with a delay of 100–500 ms and is then merged with the available syntactic completion, before it is used for GUI rendering (e.g. for emphasis of text or a popup).
- **Completion markup** may be produced by the prover in any of the following forms:

[3] To appear at `http://isabelle.in.tum.de/website-Isabelle2014` — an arbitrary snapshot as preview is provided at `http://www4.in.tum.de/~wenzelm/unofficial/UITP2014`.

- **Language context** guides the syntactic completion. Isabelle is a framework of many sub-languages, which have different requirements for completion. The language context for some text range informs the editor about the language name (e.g. to use a different keyword table), and some common flags like use of Isabelle symbols and antiquotations.
  For example, the term language in Isabelle supports symbols, but no antiquotations. In contrast, the document language (a semi-formal version of LaTeX) supports antiquotations, but no symbols. An antiquotation that puts a term inside some document source needs to switch the language context accordingly, and several such changes of language context can happen in a small piece of theory source.
  This approach already works smoothly for text that is structurally mostly correct, but a special challenge of PIDE interaction is to treat situations of partial or broken input gracefully. The expectation of the user versus the system may disagree about the intended structure of some unfinished text.
- **Completion items** result from failed name space lookups of formal entities (type names, term constants, fact names etc). Luckily the prover already has a mostly uniform concept of name spaces, in order to intern names given by the user to the actual formal entities from the context. The error situation has been slightly modified to include a list of alternative names into the error message, as PIDE markup that is not immediately visible, but available to the completion mechanism.
  For performance reasons, it is important to produce completion items only for failed name-space lookups, which are relatively rare, and not for the majority of successful ones. There is nonetheless a simple way for the user to request more information: adding a suffix of underscores to a partial name provokes an error with extra completion information. A double underscore on its own serves as wildcard to query the whole name space, but output is always truncated to a reasonable limit for display. Explicit completion requests via underscores are particularly important for the term language, because undeclared constants alone are accepted as free variables, without any error nor completion information.
- **No-completion zones** enable the prover to *negate* already discovered syntactic completions of the editor. Such non-monotonic change of the meaning of incremental document content is always critical, and can lead to erratic behaviour. Here it should be seen as a feature of last resort, to suppress odd effects when Isar keywords like "`:`" and "`|`" should remain like that, and not be offered as completion candidates for symbols "$\in$" and "$\vee$".

Spell-checking is another application of the same PIDE infrastructure, which is somewhere in between syntactic and semantic completion. Based on prover markup for the language context, e.g. to determine ranges of prose text inside document sources or comments, the editor uses a conventional dictionary-based spell-checker to propose alternatives to words spotted in the text. This is important to write books and papers based on Isabelle theory sources, which is in fact the most relevant practical application of Isabelle for 15 years.

Figure 3 illustrates spell-checking within informal text: the default dictionary does not know about `Hilbert's`, but this is not an error, merely highlighting. Moreover there is semantic completion within the term language, using an extra underscore to let the prover expose constants from the theory context.

3.2 Editor navigation

PIDE document content consists of sources that are augmented by semantic markup from the prover, as explained in [6]. The abstract syntax for the markup follows untyped XML, and the semantics is often

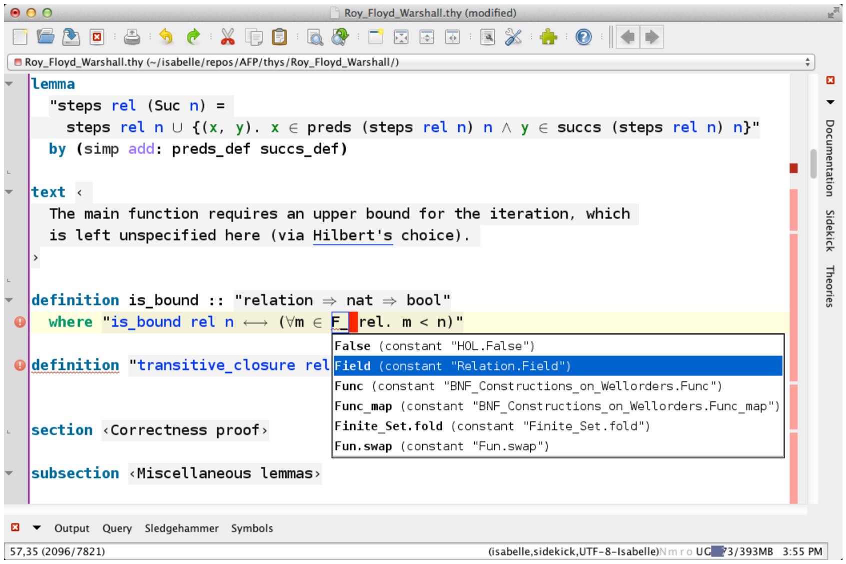

Figure 3: Spell-checking within informal text and semantic completion within terms.

close to hypertext, with occurrences of formal entities in defining or referencing positions. Thus it is rather obvious to think of standard XML/HTML rendering and browsing of PIDE documents.

In fact, early versions of Isabelle/jEdit (from 2010 to 2012) were using a basic HTML4 rendering engine, always with the anticipation for the HTML5 browser component that was promised by Sun, and delivered at last by Oracle for Java 7. None of this is used in Isabelle/jEdit today, because it introduces more problems than it solves: HTML is a very complex collection of standards in many versions and different implementations. Professional Web designers (and their tools) know how to cope with major browsers, but exotic HTML components for Java/Swing or JavaFX can hardly be expected to achieve professional quality.

Already since 2013, Isabelle/jEdit uses a slightly augmented version of the main jEdit text area, with specific support for *active areas*. Hyperlinks are an important special case of that: prover markup is turned directly into familiar clickable spots in the text (via mouse hovering with the CONTROL or COMMAND modifier key pressed). In 2014 the visual appearance approximates that of major Web browsers further, e.g. due to change of the mouse pointer. There is now also the long missing connection to an existing *Navigator* plugin from the jEdit repository, which is a rare case of successful re-use of software components. No special tricks nor reconsideration of the underlying concepts were required, to make Isabelle/jEdit converge with regular HTML browsers in this respect.

Isabelle/jEdit text areas with markup and hyperlinks are used uniformly wherever that makes sense: for the main editor buffer, output panels, tooltips etc. The user who sees a printed term somewhere can follow the implicit links to the definitions of the formal entities shown there, and return easily to the original editor location via the now standard *Back* button in the toolbar (see also figure 3 and figure 4).

3.3 Auxiliary files within the document-model

Ultimately, the main job of an IDE is to manage a collection of sources and the results of processing them seamlessly, taking implicit and explicit structural dependencies into account. So far the PIDE document model was based on two levels in the structural hierarchy: an acyclic graph of *document nodes* (theories), where each node consists of a list of *command spans* (like in Proof General [1]).

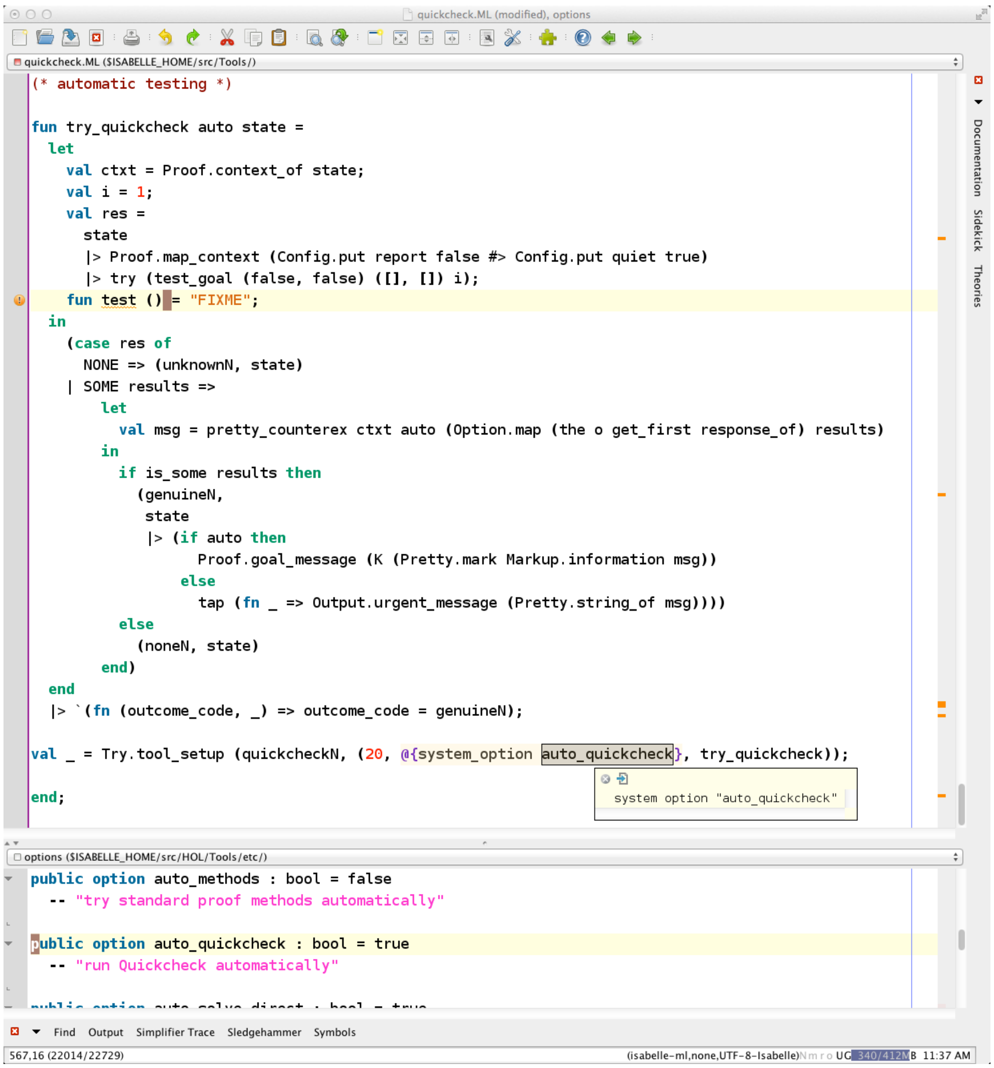

Figure 4: Live editing and browsing of Isabelle/HOL ML files

Apart from that, it was always possible to refer to auxiliary files as a semi-official feature addition, but with limited management in the IDE. That unsatisfactory situation has ended, and there is now first-class support for auxiliary files that appear as arguments to special *load commands* inside document nodes. Thus the source text is conceptually extended by so-called *text chunks* that are stored elsewhere, and may be edited / loaded / saved independently of the theory itself. The Prover IDE takes care to forward the correct version of auxiliary file content to the prover as a *blob*, but without using the global file-system.[4]

This extra file management is particularly relevant for development of Isabelle/HOL itself within

[4] By-passing the file-system is an important PIDE principle to avoid statefulness and restricting the document-model to a single version. Here the jEdit buffer management takes over this role: the current editor content is propagated to the prover as latest version, while further changes may follow.

the Prover IDE. According to usual practice of LCF-style proof assistants, the main logical environment emerges by alternating theory specifications with ML modules. It is now possible to use Isabelle/jEdit to explore the inner workings of Isabelle/HOL modules and their dependencies on theory content, notably in conjunction with the navigation support explained above (§3.2).

For live editing of the Isabelle/HOL sources, the logic session image of Isabelle/jEdit needs to be set to `Pure`, which requires a restart of the Prover IDE. Then any of the theories may be opened, e.g. `$ISABELLE_HOME/src/HOL/HOL.thy` by using that path notation literally in the jEdit file browser (even on MS Windows). The **ML_file** commands in such theories refer to Isabelle/ML modules that are compiled on the spot. By default the prover reads the source from the file-system, but by following the implicit hyperlink of the file argument (or opening files in the jEdit file-browser) the editor takes over the responsibility for the sources and its subsequent changes.

Thus the user may edit Isabelle/ML source files, without ever saving the content, while the Poly/ML compiler provides continuous feedback on warnings, errors, name references, inferred types etc. as part of the PIDE document model (see figure 4). This works reasonably well for source files up to 100 KB each. The total volume of ML sources contributing to Isabelle/HOL is actually so high that its cumulative PIDE markup requires more than 2 GB Java heap space. This performance bottle-neck is addressed by some special tricks with asynchronous print functions (§2) which were introduced in 2013 for quite different applications. Here the mechanism is re-used as follows: Poly/ML compiler markup is stored in compact form within the ML process, and only reported to the editor when the corresponding ML file becomes part of the visible perspective. The document markup is removed from the editor process when visibility gets lost.

Thus the massive amount of PIDE markup produced by the ML compiler is "swapped-in" and "swapped-out" on demand, without changing the content of the ML environment itself. Consequently the full Isabelle/HOL bootstrap environment can be edited with full Poly/ML markup, even on small computers with only 4–8 GB memory.

As a corollary to this scalable approach to continuous editing and compilation of Isabelle/ML files, there is also support for official Standard ML via the **SML_file** command. Thus Isabelle/jEdit can be used as IDE for SML'97, without any connection to theory or proof development. The two ML environments are managed independently within the same runtime system, but there are also simple means to exchange toplevel ML bindings, e.g. to re-use the parallel functional programming library of Isabelle/ML in Standard ML, or to print messages in Standard ML that are recognized by the Prover IDE for its *Output* panel. The Prover IDE provides some simple examples for that in its *Documentation* panel, in the entry for `$ISABELLE_HOME/src/Tools/SML/Examples.thy` that is only a single click away from direct editing and browsing. Here the enclosing theory is used like a project file for SML modules, with the possibility to add extra explanations around it.

Thus Standard ML has gained a reasonably modern IDE after some decades of waiting. Navigation of the sources works as usual (§3.2), but semantic completion (§3.1) is still missing because Poly/ML (version 5.5.2) does not provide that information. Also lacking is support for the interactive debugger of Poly/ML. Such further fine-grained interaction with the ongoing execution process would be quite useful for other Isabelle languages as well, e.g. to analyse the behaviour of tactical expressions beyond the single-stepping of outermost commands or adhoc printing of intermediate states.

If that train of thought is continued further, it could meet with recent trends towards "live programming", as advocated by Bret Victor on his blog[5], for example. This is a revival and continuation of older

[5] `http://worrydream.com`

ideas from Smalltalk of the 1970s and 1980s, but adapted to the possibilities of the hardware from today. Interactive theorem proving has always been conceptually close to such "dynamic" development models, and the removal of the TTY loop in Isabelle/PIDE could help to demonstrate that in reality.

A major difference to Smalltalk and classic object-oriented programming is that Isabelle/PIDE document content is immutable and processed monotonically, in a timeless and stateless manner, with functional update of pure data. That used to be costly in the past, but on the multicore hardware of today immutability is a big asset.

References

[1] David Aspinall (2000): *Proof General: A Generic Tool for Proof Development*. In Susanne Graf & Michael Schwartzbach, editors: *European Joint Conferences on Theory and Practice of Software (ETAPS)*, *LNCS* 1785, Springer, doi:10.1007/3-540-46419-0_3.

[2] Christoph Lüth & Martin Ring (2013): *A Web Interface for Isabelle: The Next Generation*. In Jacques Carette et al., editors: *Intelligent Computer Mathematics (CICM 2013)*, *LNCS* 7961, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-39320-4_22`.

[3] Christoph Lüth & Martin Ring (2014): *Collaborative Interactive Theorem Proving with Clide*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving (ITP 2014)*, *LNCS* , Springer.

[4] L. C. Paulson, T. Nipkow & G. Klein, editors: *Archive of Formal Proofs*. `http://afp.sourceforge.net/`.

[5] M. Wenzel (2009): *Parallel Proof Checking in Isabelle/Isar*. In G. Dos Reis & L. Théry, editors: *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*, ACM Digital Library.

[6] M. Wenzel (2011): *Isabelle as Document-oriented Proof Assistant*. In J. H. Davenport, W. M. Farmer, F. Rabe & J. Urban, editors: *Conference on Intelligent Computer Mathematics / Mathematical Knowledge Management (CICM/MKM 2011)*, *LNAI* 6824, Springer.

[7] Makarius Wenzel (2010): *Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit*. In C. Sacerdoti Coen & D. Aspinall, editors: *User Interfaces for Theorem Provers (UITP 2010)*, ENTCS.

[8] Makarius Wenzel (2012): *Isabelle/jEdit — a Prover IDE within the PIDE framework*. In J. Jeuring et al., editors: *Conference on Intelligent Computer Mathematics (CICM 2012)*, *LNAI* 7362, Springer, doi:10.1007/978-3-642-31374-5_38.

[9] Makarius Wenzel (2013): *Isabelle/jEdit*. Part of Isabelle distribution. `http://isabelle.in.tum.de/website-Isabelle2013-2/dist/Isabelle2013-2/doc/jedit.pdf`.

[10] Makarius Wenzel (2013): *READ-EVAL-PRINT in Parallel and Asynchronous Proof-checking*. In Cezary Kaliszyk & Christoph Lüth, editors: *User Interfaces for Theorem Provers (UITP 2012)*, *EPTCS* 118. Available at `http://dx.doi.org/10.4204/EPTCS.118.4`.

[11] Makarius Wenzel (2013): *Shared-Memory Multiprocessing for Interactive Theorem Proving*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *Interactive Theorem Proving — 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, *Lecture Notes in Computer Science* 7998, Springer. Available at `http://dx.doi.org/10.1007/978-3-642-39634-2_30`.

[12] Makarius Wenzel (2014): *Asynchronous User Interaction and Tool Integration in Isabelle/PIDE*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving — 5th International Conference, ITP 2014, Vienna, Austria*, *Lecture Notes in Computer Science* 8558, Springer.