Isabelle/PIDE after 10 years of development

Makarius Wenzel
https://sketis.net

The beginnings of the Isabelle Prover IDE framework (Isabelle/PIDE) go back to the year 2008. This is a report on the project after 10 years, with system descriptions for various PIDE front-ends, namely (1) Isabelle/jEdit, the main PIDE application and default Isabelle user-interface, (2) Isabelle/VSCode, an experimental application of the Language Server Protocol in Visual Studio Code (by Microsoft), and (3) a headless PIDE server with JSON protocol messages over TCP sockets.

1 Introduction

Around 2008/2009 the initial efforts for multicore support in Isabelle/ML had reached a certain point of stability and scalability [4, 2], so I started to investigate the combination of parallel processing with user interaction. This eventually lead to a document-oriented model of asynchronous editing under the label of Isabelle/PIDE [5, 7, 9, 8]. The main ideas of the PIDE approach are as follows:

- The **prover** supports asynchronous *document edits* and *markup reports* natively. The old read-eval-print loop is discontinued: protocol commands (like `Document.update`) manage document changes, while regular prover commands (like **definition**, **theorem**, **proof**, **qed**) are absorbed into the document as data.
- The **editor** turns *physical edits* on its open buffers into *mathematical updates* of the document model: here is the boundary of the stateful GUI wrt. the timeless / stateless document model. For rendering of text views, the editor retrieves accumulated markup as is available on the spot: that happens frequently and needs to finish within milliseconds. Rendering is an approximation of continued processing by the prover, and later GUI updates improve upon the partial view until it converges to a consolidated state.
- Add-on **tools** may participate in proof document processing and markup reports, e.g. automated provers or disprovers that attach extra markup or helpful *information* messages. Tools are usually started and stopped after a timeout of a few seconds, and are treated as closed function invocation without further interaction (apart from interruption of an abandoned attempt).
- The **user** has the main authority (and responsibility) to construct proof documents, but is assisted by the GUI rendering of cumulative PIDE markup (e.g. clickable suggestions for proof templates).

In other words, PIDE is an *orchestration* of prover, editor, and add-on tools to enable the user to compose proof documents. The interplay of all this depends on real-time constraints and timing parameters, so PIDE is more like a computer game rather than a text editor. This approach ultimately challenges the ITP community to introduce genuine prover interaction, to go significantly beyond copy-paste between an editor and a prover process (as we have seen for vi or Emacs in the past).

Isabelle/PIDE got quite far in that respect in the past 10 years, but many problems had to be overcome: conceptual, technical, and social (some users were strongly attached to Emacs). October 2014 was a notable point-of-no-return, when the last remains of the Isabelle read-eval-print loop were removed, together with its wrapper for Proof General Emacs [1]. That was once important for Isabelle, but today

Submitted to:
UITP 2018

Proof General is only used for Coq and its generality is now lost, despite historic references to other provers.[1]

The Isabelle/PIDE implementation is notable for bridging the gap between Isabelle/ML and the Java world thanks to the Isabelle/Scala library for Isabelle system programming: the total size of Isabelle/Scala sources has grown over 10 years to approximately 1.5 MB, with a rather compact programming style similar to Isabelle/ML. There is an internal PIDE protocol on a bidirectional byte-channel to connect the ML and Scala process: details are explained in [6] by the example for Coq as experimental PIDE back-end. It is also possible to define external protocols to connect to Isabelle/Scala from the outside (e.g. see §3 and §4).

2 Isabelle/jEdit

Isabelle/jEdit (figure 1) is the best-developed application of Isabelle/PIDE and the default user-interface for Isabelle. It is distributed on the Isabelle website `https://isabelle.in.tum.de` (and mirrors) as a fully-integrated application for Linux, Mac OS X, and Windows: users merely need to download, unpack, and run it. It is one big *portable application*, which bundles all non-trivial requirements: Java, Scala, Poly/ML, add-on tools and libraries. The only external dependencies are standard C/C++ libraries and Unix tools like GNU Bash and Perl.[2]

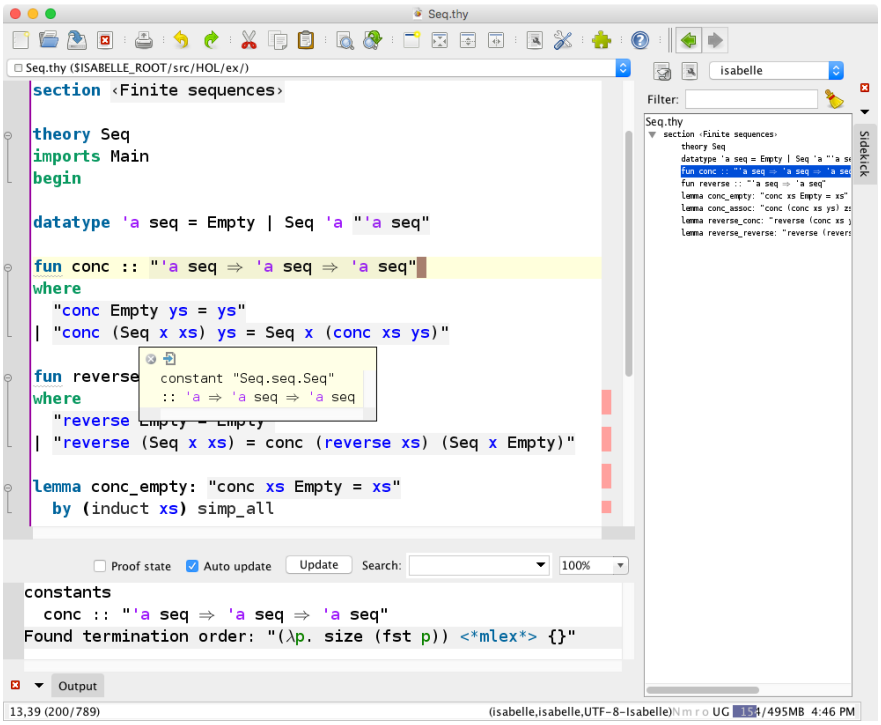

Figure 1: The Isabelle/jEdit Prover IDE

[1] `https://proofgeneral.github.io`

[2] On Windows that Unix environment is provided by a bundled Cygwin installation, which was once more important than today. Its remaining purpose is system glue for `etc/settings` Bash scripts and `SIGINT` emulation on external processes. Most other tools — Java, Poly/ML, external provers — are now native Windows executables.

The official 1.0 release of Isabelle/jEdit was published with Isabelle2011 (October 2011) — still somewhat experimental, but already usable for small theory developments. The major version number has been incremented for each Isabelle release, so Isabelle2018 (August 2018?) will include Isabelle/jEdit version 10.0. In recent years, Isabelle/jEdit has become a *filthy rich client* that requires 4–8 GB memory and 1–2 CPU cores to get started with small examples. These resources should be doubled for bigger applications, e.g. the `HOL-Analysis` library (figure 2 on page 10) with its massive material ported from HOL Light.

The Isabelle/jEdit manual [10], which is also accessible in the GUI panel for *Documentation*, explains the main concepts and features of the IDE in more than 60 pages. Here is a brief overview roughly in the order of the manual:

- Near-native GUI look-and-feel on Linux (GTK+), Mac OS X, and Windows, with good support for very high resolution ("4K", "Ultra HD", "HiDPI", "Retina").
- Application fonts for reliable rendering of Isabelle symbols: mathematical symbols similar to LaTeX, document markup / markdown symbols, icons for files / directories / documents, control symbols for superscript / subscript / bold face.
- Input methods for Isabelle symbols, via ASCII abbreviations, completion popup, or GUI panel.
- Automatic indentation according to syntactic structure (e.g. block structure of Isar proofs), and semantic information (e.g. number of subgoals in unstructured proof scripts).
- Tree views via the SideKick plugin of jEdit: Isabelle document outline (section headings), Isabelle context (**locale**, **class**, **context** structure), raw markup tree for document source.
- Text folding according to document structure (e.g. block structure of Isar proofs).
- Management of document file dependencies, based on theory **imports**, auxiliary files (e.g. **ML_file**, **SML_file**) and session specifications of cumulative `ROOT` and `ROOTS` files from Isabelle project directories.
- Output panel for explicit messages (including warnings and errors), and optional proof state.
- State panel for explicit proof state output.
- Query panel for interactive exploration of the the formal context: find theorems, find constants, print context.
- Tooltips and hyperlinks that work uniformly for input sources and output text, including recursive output of tooltips. This allows to explore the full depth of formal content, with nested tooltip windows stacking up in the GUI.
- Highlighting formal scopes in the source text: binding and referencing positions of constants, variables, etc. There is also minimal support to select and rename corresponding identifiers, but no formal "refactoring" yet.
- Completion of syntax keywords, Isabelle symbols, and various semantic entities: variables, type names, term constants, fact names etc., but also session-qualified theory names, file-system paths, spell-checker dictionary entries, BibTeX database entries (for `.bib` files that happen to be open in the editor).
- Automatically tried tools that operate on input source asynchronously, without blocking the user. Notably: proposal of automatic proof methods, counter-examples via Quickcheck and Nitpick, heavy automated reasoning via Sledgehammer, trivial fact application via Solve Direct. All tools

are subject to a start and stop timeouts of a few seconds, to avoid wasting CPU resources while the user is still busy editing. Heavy tools like Nitpick or Sledgehammer are disabled by default.

- Sledgehammer panel as a GUI front-end to the **sledgehammer** command.
- Basic support for Isabelle document preparation: outline with section headings, markdown structure of lists (itemize, enumeration, description), adhoc management of BibTEX database files and citations, minimalistic HTML preview via builtin HTTP server and external browser.
- Isabelle/ML source outline via standard conventions for ML comments as section headings.
- Isabelle/ML source-level debugger within the IDE.
- Timing panel for run-time measurements of prover commands.
- Monitor panel for Isabelle/ML and Poly/ML performance data: future tasks, worker threads, garbage collection, heap size, low-level threads, cumulative CPU time and relative speed on multiple CPU cores.
- Scala console for command-line interaction with the running Isabelle/Scala/JVM environment.

IDE support for Isabelle development. There is still no IDE support for Isabelle/Scala/PIDE development itself: I simply do not understand how the Scala compiler is managed at run-time, its internals are very "object-oriented" and thus rather complex. I usually edit Scala sources in the Scala mode of regular jEdit (not Isabelle/jEdit), and run `isabelle jedit -b` on the Unix command-line to rebuild it.

In contrast, Isabelle/ML compilation (and debugging) has been properly integrated into the evaluation model of the PIDE document. Incremental ML compilation at run-time works nicely due to `PolyML.Compiler` APIs provided by David Matthews, who is the master of Poly/ML[3]. The toplevel ML environment is managed within the Isabelle theory context, and thus as stateless as other logical declarations. In order to follow the bootstrap process of Isabelle/HOL, one merely needs to open its sources (e.g. at the entry points `$ISABELLE_HOME/src/HOL/HOL.thy` or `$ISABELLE_HOME/src/HOL/Main.thy`) using `isabelle jedit -l Pure` and then work through hundreds of automatically loaded files.

The self-application of the Prover IDE also covers the Isabelle/Pure bootstrap process: by loading `$ISABELLE_HOME/src/Pure/ROOT.ML` into Isabelle/jEdit it is treated like a theory file in the initial ML environment and its **ML_file** commands are processed in the given order, to build a *virtual* Isabelle/Pure within the *real* Isabelle/Pure of the running PIDE session. In the original bootstrap process from bare Poly/ML, `ML_file` is just an alias for the traditional ML function `use`. IDE support for Isabelle/Pure has become indispensable for further development, but also helps users to understand the sources thanks to formal markup (e.g. inferred types, binding scopes and hyperlinks for ML identifiers). For example, ML definitions of prover commands can be easily followed using hyperlinks in example theories; their ML definitions become "active" after the file `$ISABELLE_HOME/src/Pure/ROOT.ML` has been opened.[4]

Likewise, any session `ROOT` file (e.g. `$ISABELLE_HOME/src/HOL/ROOT`) is treated like as a special theory with a dummy Isar command **session** that provides some formal markup for its arguments (e.g. hyperlinks to imported sessions and theories). This gives an approximative semantic view of session specifications that are normally processed in batch-mode only. It also demonstrates how to implement IDE add-ons with minimal effort (see also `Sessions.command_parser` in Isabelle/ML).

[3] http://polyml.org

[4] This additional requirement also serves as a guard to prevent processing of Isabelle/Pure sources, when users jump to the definition of a prover command by accident.

3 Isabelle/VSCode

Visual Studio Code (or VSCode) is an open-source project by Microsoft, with the slogan "*Code editing. Redefined. Free. Open source. Runs everywhere.*"[5] It is a cross-platform application for Linux, Mac OS X, and Windows — based on the Electron framework[6], which combines the Node.js engine with the Chromium browser. The idea of VSCode is to augment text editing with semantic checking as seen in high-end IDEs, but to make the overall experience more smooth and light-weight. There is a public *Marketplace*[7] of extensions for the VSCode editor with add-ons for many languages.

Isabelle/VSCode (figure 3 on page 11) is an extension for VSCode that uses the Isabelle/PIDE programming interface to provide a semantic languages service. It consists of a small TypeScript module within VSCode and an external Isabelle/Scala process that runs the *Language Server Protocol* (a subset of version 3.2)[8]. The protocol is based on JSON-RPC: I have added sufficient JSON support to Isabelle/Scala to make this an easy exercise, it can be reused for other protocols (e.g. §4). Isabelle/VSCode is already part of the Isabelle distribution, in Isabelle2017 (October 2017) it was officially released as version 1.0. Here is a summary of its main features:

- Static syntax tables for Isabelle `.thy` and `.ML` files.[9]
- Implicit dependency management of sources, subject to changes of theory files within the editor, as well as external file-system events (updates on file-system content are treated like edits).
- Implicit formal checking of theory files, using the *cursor position* of the active editor panel as indication for relevant spots.[10]
- Text overview lane with formal status of prover commands (unprocessed, running, error, warning).
- Prover messages within the source text (errors/warnings and information messages).
- Semantic text decorations: colors for free/bound variables, inferred types etc.[11]
- Visual indication of formal scopes and hyperlinks for formal entities.
- Implicit proof state output via the VSCode message channel *Isabelle Output*.
- Explicit proof state output via separate GUI panel (command `isabelle.state`).
- HTML preview via separate GUI panel (command `isabelle.preview`).
- Rich completion information: Isabelle symbols (e.g. `\forall` or ∀), outer syntax keywords, formal entities, file-system paths, BibTEX entries etc.
- Spell-checking of informal texts, including dictionary operations: via the regular completion dialog.

[5] https://code.visualstudio.com

[6] https://electronjs.org

[7] https://marketplace.visualstudio.com/VSCode

[8] https://microsoft.github.io/language-server-protocol/specification

[9] The latter causes a conflict with `.ml` files for OCaml: VSCode file-extensions are case-insensitive — presumably a legacy feature of MS-DOS.

[10] VSCode lacks APIs to access the official editor view port, which is used a lot in Isabelle/jEdit to guide implicit document processing.

[11] This uses private PIDE extensions for the Language Server Protocol. The protocol implementation in VSCode makes it very easy to invent new messages, although the official protocol specification does not cover extensions.

Running Isabelle/VSCode requires manual configuration of VSCode to activate the Isabelle extension[12] and tinker with various settings — the official download counter indicates that a few hundred users have already tried that for Isabelle2017. Since the center of this universe is VSCode and not Isabelle, a few things can go wrong (just like Emacs configuration in the old times of Proof General). Changing versions of Node.js modules may also break the application: this is a highly dynamic ecosystem.

Isabelle/VSCode 1.0 may serve as demonstration that Isabelle/PIDE and VSCode can work together reasonably well, but it can hardly compete with Isabelle/jEdit 9.0 in current Isabelle2017 or 10.0 in the coming Isabelle2018 release. I have made only small changes to Isabelle/VSCode after Isabelle2017, so in Isabelle2018 it will probably be published as version 1.1 (instead of 2.0).

Future potential of the VSCode platform. Isabelle/jEdit with its venerable jEdit text editor and vintage Java/Swing platform is well-established, but hardly moving forwards anymore: it is somewhat stagnant at a high level of sophistication.

In contrast, the VSCode platform with its Node.js and Chromium basis is much more dynamic, but a lot of work still needs to be done. It is rather typical that many VSCode extensions are mere experiments at version 0.x or 0.0.x, and the main editor platform is still evolving (with new releases every month).

The very idea to use untyped JavaScript and dynamic Node.js might be distasteful to statically-typed ML people, but VSCode is mainly written in TypeScript and there is also Scala.js[13] that could eventually help to migrate parts of Isabelle/PIDE to the JavaScript platform.

VSCode should be seen as a hybrid of text editor and HTML browser. This could open possibilities to high-quality rendering of Isabelle proof documents with HTML5/CSS, while the user is editing the formal sources. A proof-of-concept for HTML preview is shown in figure 4 on page 12: it works like the existing Markdown preview of VSCode. The generated HTML is produced by Isabelle/Scala in the same manner as the traditional HTML presentation of theories[14]. It is presently rather crude and merely imitates Isabelle/jEdit syntax highlighting to some extent, but any improvements on the Isabelle/Scala side will immediately impact Isabelle/VSCode as well — recall that it is ultimately a Chromium web browser wrapped in a funny way.

4 Headless PIDE server

Isabelle/PIDE has abolished the traditional prover command-line in October 2014, but the coming Isabelle2018 release will include a PIDE server as a replacement, via `isabelle server` on the Unix command-line, or via `isabelle.Server.init()` in Isabelle/Scala. The server imitates a "terminate-stay-resident" desktop application, but there is no GUI front-end, so it is "headless". It uses line-oriented protocol messages with JSON syntax by default; it is also possible to use the internal YXML message format of PIDE, but there are presently no protocol commands for that.

Each user can run multiple server processes, with at most one instance for each server name; the global state is maintained in `$ISABELLE_HOME_USER/servers.db` as an SQLite database. Each server process can manage multiple logic *sessions* and concurrent tasks to use *theories*. Sessions are identified by random UUIDs[15] and may be shared by server connections if desired.

[12] https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2017
[13] https://www.scala-js.org
[14] http://isabelle.in.tum.de/dist/library/index.html
[15] https://tools.ietf.org/html/rfc4122

The client/server arrangement via TCP sockets also opens possibilities for remote Isabelle services that are accessed by local applications, e.g. via an SSH tunnel, but there is no specific support yet (in Isabelle/607957640057 from the repository).

Protocol messages. The Isabelle server listens on a regular TCP socket, using a line-oriented protocol of structured messages. Input *commands* and output *results* (via `OK` or `ERROR`) are strictly alternating on the toplevel, but commands may also return a *task* identifier to indicate an ongoing asynchronous process that is joined later (via `FINISHED` or `FAILED`). Asynchronous `NOTE` messages may occur at any time: they are independent of the main command-result protocol.

For example, the synchronous `echo` command immediately returns its argument as `OK` result. In contrast, the asynchronous `session_build` command returns `OK {"task":`*id*`}` and continues in the background. It will eventually produce `FINISHED {"task":`*id*`,...}` or `FAILED {"task":`*id*`,...}` with the final result. Intermediately, it may emit asynchronous messages of the form `NOTE {"task":`*id*`,...}` to inform about its progress. Due to the explicit task identifier, the client can show these messages in the proper context, e.g. a GUI window for this particular session build job.

Server commands. The following server commands are implemented in recent Isabelle repository versions (e.g. Isabelle/607957640057 from the repository)[16]:

- `help`: return the list of server command names.
- `echo`: return the command argument as regular result (`OK`). This is the identity function for protocol messages.
- `shutdown`: force a shutdown of the connected server process.
- `cancel`: attempt to cancel a specified asynchronous task (always succeeds but may have no effect).
- `session_build` (asynchronous): prepare a session image for interactive use of theories. This is a restricted version of the command-line tool `isabelle build`. It avoids the overhead of creating a fresh Isabelle/Scala/JVM process for each build task.
- `session_start` (asynchronous): start a new Isabelle/PIDE session with underlying Isabelle/ML process, based on a session image that is produced on demand using `session_build`. This resembles `isabelle jedit -l` *session*, without the GUI environment around it.
- `session_stop` (asynchronous): force a shutdown of the identified PIDE session.
- `use_theories` (asynchronous): update the identified session by adding the current version of theory files to it, resolving theory imports implicitly. The command succeeds eventually, when all theories have been *consolidated* in the sense that the outermost command structure has finished (or failed) and the final **end** command of each theory has been checked. This resembles the node status in the *Theories* panel of Isabelle/jEdit, where a thick rectangle indicates a consolidated state.
- `purge_theories` (asynchronous): updates the identified session by removing theories that are no longer required. Note that "garbage collection" of loaded theories is not automatic, because a client might want to reuse already loaded theory imports later on.

[16] See `https://isabelle.sketis.net/devel/release_snapshot` for nightly snapshots, notably the included **system** manual chapter 4.

The main command of the server is `use_theories`, it provides an analogue to traditional `Thy_Info.use_theories` in Isabelle/ML, but with full concurrency and Isabelle/PIDE markup. It also avoids repeated startup and shutdown of Isabelle sessions, which can take several seconds.

The `use_theories` refers to a collection of theories via the local file-system (usually a temporary directory), lets the prover process them, and returns consolidated results (with warnings and errors). The default output imitates an old-fashioned TTY with a fixed-width font at 76 characters per line. Behind that are the same pretty-trees with PIDE markup, as seen in the Isabelle/jEdit front-end. A refined version of the server might expose that information as YXML trees, but the client would have to do its own rendering (calculating breaks according to the font metrics).

5 Conclusion

After 10 years of development, Isabelle/PIDE has come quite far, but is also far from finished. Here are some important lines of future development:

- Even more scalability of the Isabelle/jEdit front-end (and underlying PIDE back-end infrastructure). The ultimate goal is to load the whole *Archive of Formal Proofs*[17] into a single IDE session, but that is growing at a high rate (quoting [9] from 4 years ago, when AFP was about half the size of today). See also my recent blog entry (with paper) about scaling of Isabelle technology[18].
- More serious support for Isabelle/VSCode. In particular, it should be distributed as a robust standalone application like Isabelle/jEdit, even if it might annoy people who like to tinker with VSCode extensions and options by themselves.
- Further elaboration of the headless PIDE server, e.g. support for SSH tunneling (which is already available in Isabelle/Scala) and further moves towards a local IDE front-end with remote PIDE session back-end.

The last point is again about scaling (up and down):

scaling up the remote server side, with a big Isabelle/ML/Scala/JVM/PIDE session running "in the cloud",

scaling down the local IDE side, with a reduced Java or JavaScript, lets say with 2 CPU cores and 4 GB memory. A more frugal browser front-end (e.g. for AFP exploration) might eventually fit into 1–2 GB (without any application download), but that is pure speculation at the moment.

Another open problem is serious PIDE support for other provers, such as Coq, HOL4, HOL Light. There have been some experiments in the past [6, 3], but these never reached end-users. I still see high-end Prover IDEs as desirable for other ITP systems, not just Isabelle, and remain open to collaborations.

References

[1] David Aspinall, Christoph Lüth & Daniel Winterstein (2007): *A Framework for Interactive Proof*. In M. Kauers, Manfred Kerber, Robert Miner & Wolfgang Windsteiger, editors: *Towards Mechanized Mathematical Assistants (CALCULEMUS and MKM 2007)*, *LNAI* 4573, Springer.

[17] https://www.isa-afp.org

[18] https://sketis.net/2017/scaling-isabelle-proof-document-processing

[2] D. Matthews & M. Wenzel (2010): *Efficient Parallel Programming in Poly/ML and Isabelle/ML*. In: *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP 2010)*.

[3] Carst Tankink (2014): *PIDE for Asynchronous Interaction with Coq*. In Christoph Benzmüller & Bruno Woltzenlogel Paleo, editors: *User Interfaces for Theorem Provers (UITP 2014)*, *EPTCS* 167, doi:10.4204/EPTCS.167.9. Available at https://doi.org/10.4204/EPTCS.167.9.

[4] 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.

[5] 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, doi:10.1016/j.entcs.2012.06.009.

[6] Makarius Wenzel (2013): *PIDE as front-end technology for Coq*. Available at http://arxiv.org/abs/1304.6626.

[7] 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, doi:10.4204/EPTCS.118.4.

[8] 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, doi:10.1007/978-3-319-08970-6_33.

[9] Makarius Wenzel (2014): *System description: Isabelle/jEdit in 2014*. In Christoph Benzmüller & Bruno Woltzenlogel Paleo, editors: *User Interfaces for Theorem Provers (UITP 2014)*, *EPTCS* 167, doi:10.4204/EPTCS.167.10. Available at https://doi.org/10.4204/EPTCS.167.10.

[10] Makarius Wenzel (2017): *Isabelle/jEdit*. Part of Isabelle distribution. http://isabelle.in.tum.de/website-Isabelle2017/dist/Isabelle2017/doc/jedit.pdf.

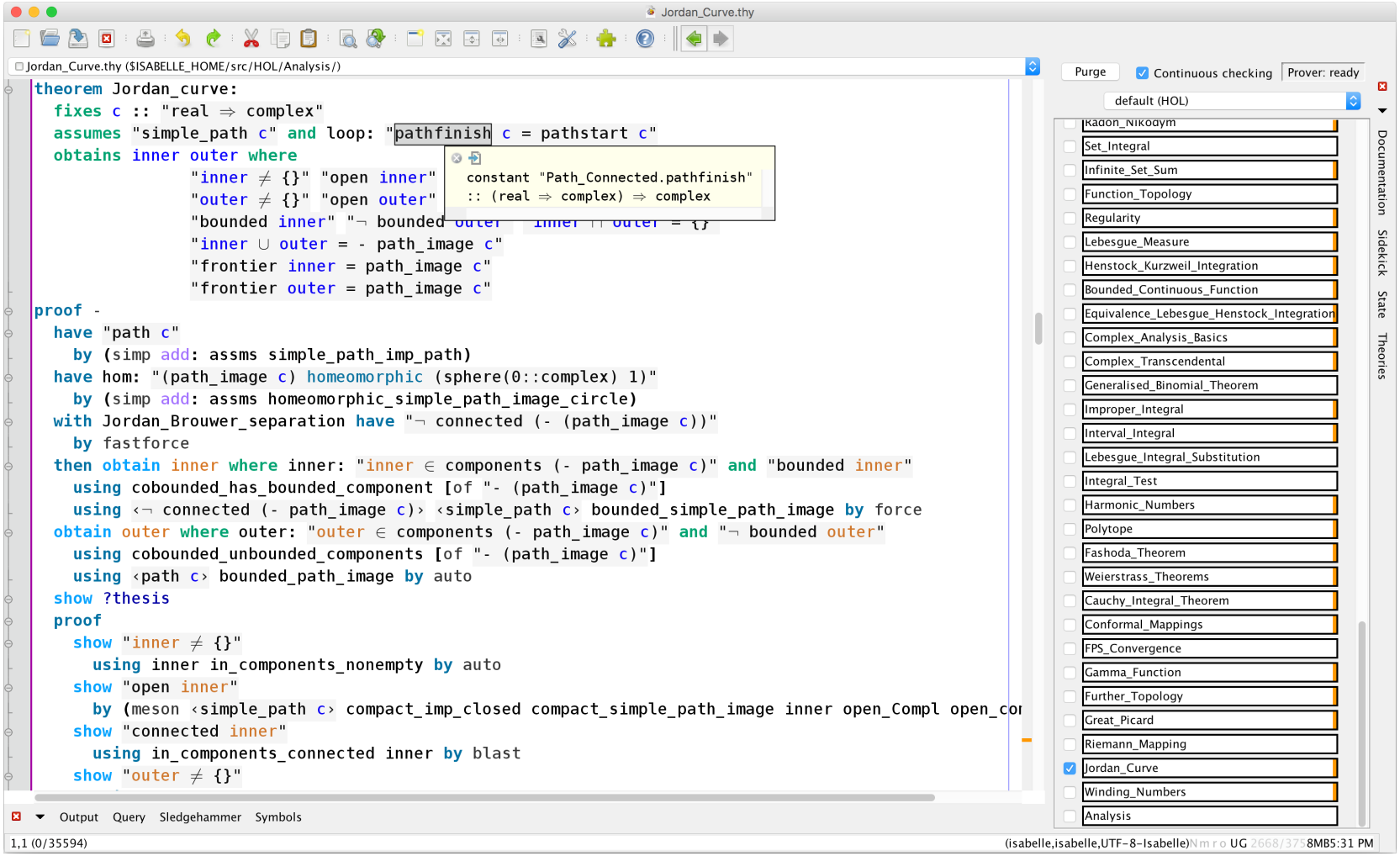

Figure 2: Session `HOL-Analysis` within Isabelle/jEdit (ML process: 2.8 GB, JVM process: 3.5 GB)

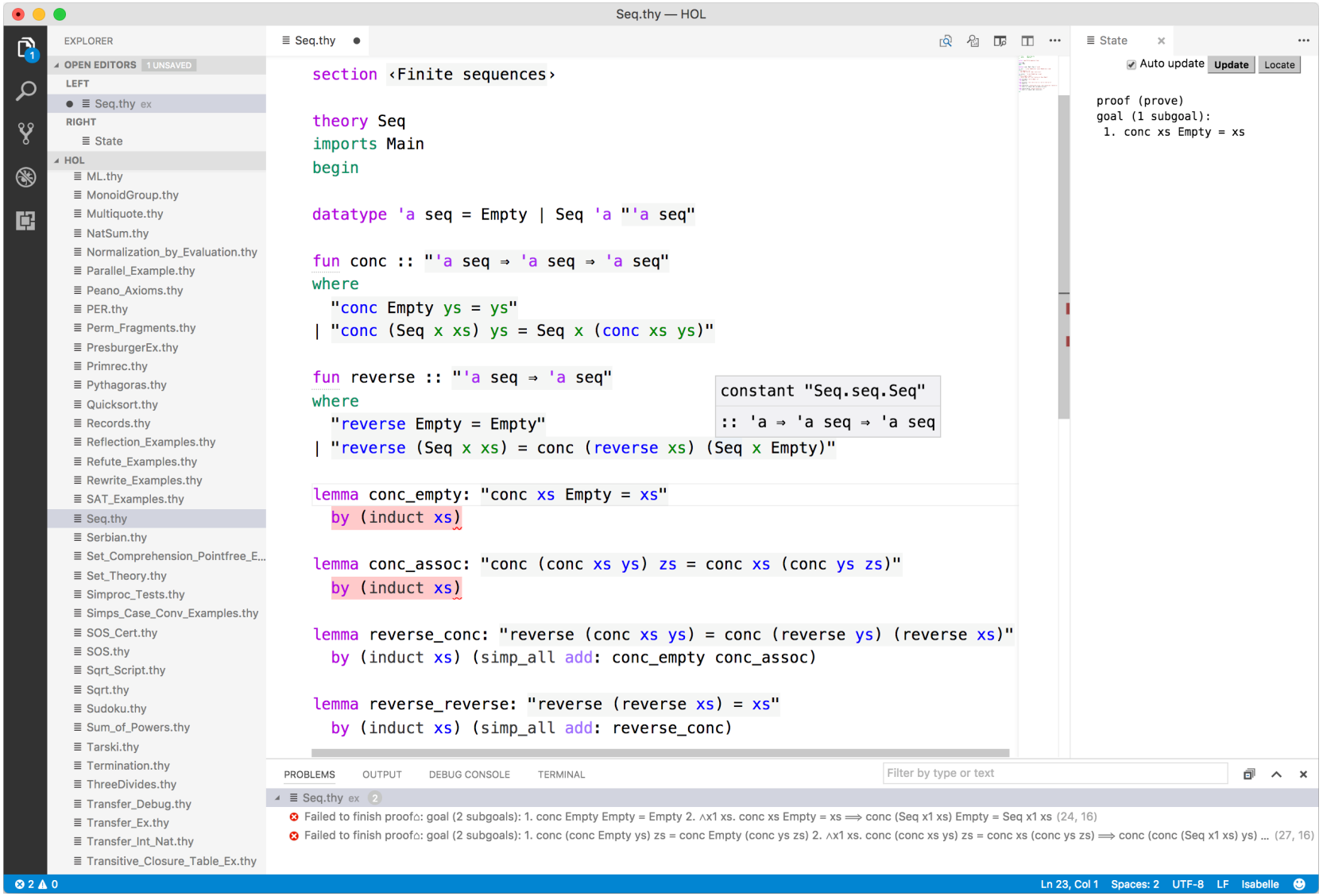

Figure 3: Isabelle/VSCode Prover IDE

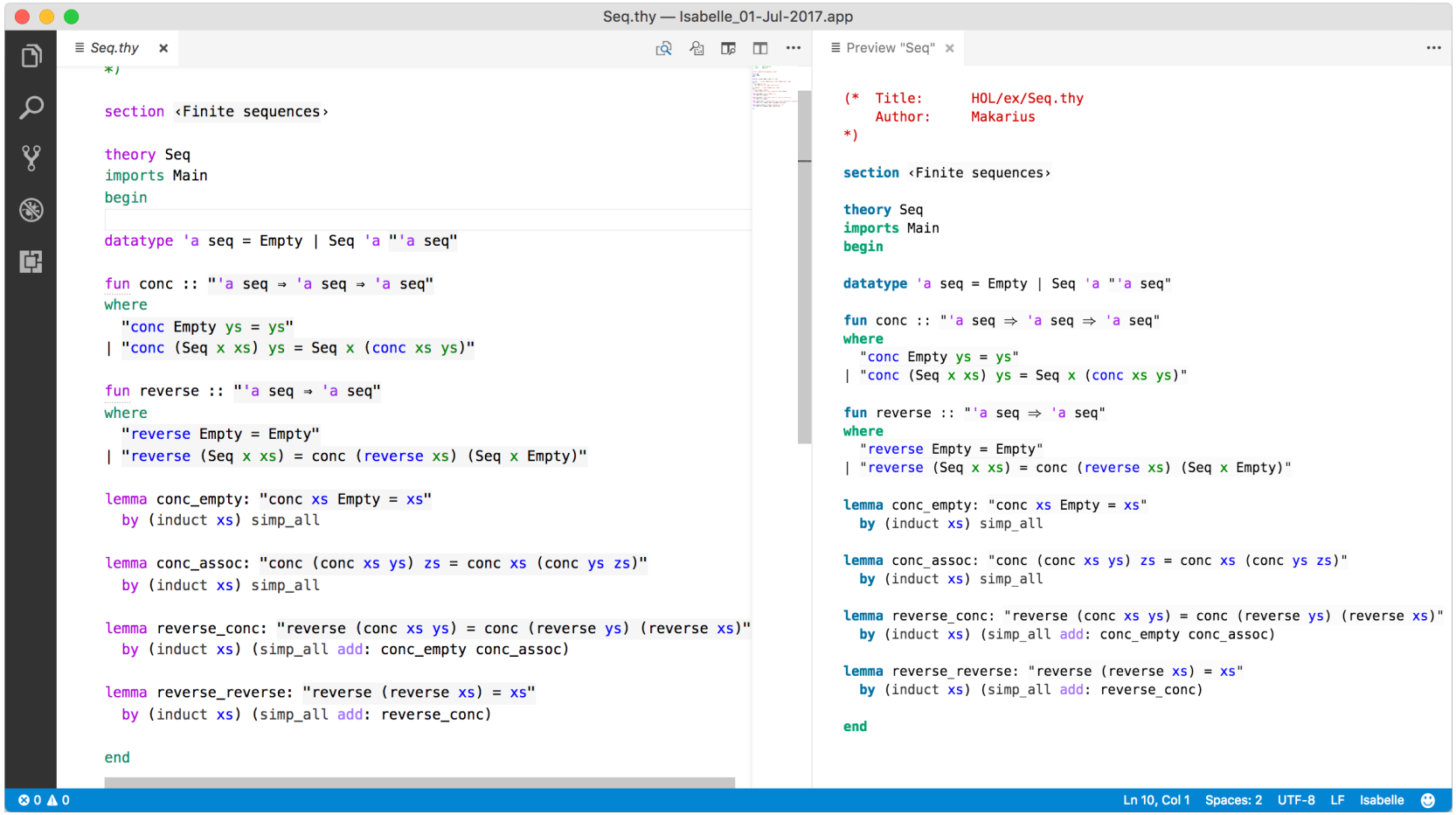

Figure 4: Isabelle/VSCode Prover IDE with built-in HTML preview (Chromium)