Scaling Isabelle Proof Document Processing

Makarius Wenzel*
December 2017

This is a study of performance requirements, technological side-conditions, and possibilities for scaling of formal proof document processing in Isabelle and The Archive of Formal Proofs (AFP). The approaches range from simple changes of system parameters and basic system programming with standard APIs to more ambitious reforms of Isabelle and the underlying Poly/ML system. The running examples for typical scalability issues are existing formalizations of classical analysis and differential equations. Such applications can be further extrapolated towards a development of Financial Mathematics (e.g. Itô calculus).

This document is based on Isabelle/74bd55f1206d and AFP/f6ca248dd250 (November 2017), with minor changes over the official release Isabelle2017 (October 2017).

Contents

*Supported by *Mu Operator GmbH, Frankfurt am Main*. Document published with permission.

1 Introduction

1.1 The Isabelle platform: formal document processing

Isabelle http://www.cl.cam.ac.uk/research/hvg/Isabelle was originally developed in 1986/1989 by L. C. Paulson as a generic *proof assistant*, for a variety of logics based on minimal Higher-Order Logic (Isabelle/Pure).

After three decades of ongoing development, the Isabelle platform has become a universal framework for *formal proof documents* that are written in *domain-specific formal languages*. E.g. symbolic λ-calculus for mathematical statements, the Isar language for formal proofs, the Eisbach language for proof methods, the Isabelle document language for informal explanations with embedded formal entities. The Isabelle/ML language is somewhat special: it is both the implementation and extension language of the formal environment; it also allows to introduce new domain-specific languages (e.g. a language of syntax diagrams with references to formal entities [4, §4.5]).

In recent years, Isabelle/Scala has become the language for *Isabelle systems programming* outside the formal environment: it manages Isabelle source files, Isabelle/ML processes and the results of formal checking; it is able to connect the pure world of formalized mathematics to the real world of user-interfaces (Prover IDE) and network services (e.g. for HTTP or JSON-RPC).

A user who downloads and runs the main Isabelle application encounters a "filthy-rich client" that is based on the full stack of Isabelle technologies, tools and languages. It presents itself as an advanced editor with semantic annotation of document sources, asynchronous checking in real-time, with parallel processing on multicore hardware.

Isabelle document sources are organized as *theory files* that typically contain:

- mathematical definitions, statements and proofs,
- informal explanations in natural language with embedded formal entities,
- executable code that is generated from specifications: functional programs in SML, OCaml, Haskell, Scala.

A collection of theory files is called *session*. Each session is derived from a *parent session*, which defines an overall *session tree* with re-use of already checked results; the `Pure` session is the root.

Here are some example sessions from the official Isabelle distribution, with linear session-parent dependencies in the following order:

1. `Pure`: the main system implementation and bootstrap environment. This is rarely used directly in applications.
2. `HOL`: classical Higher-Order Logic with many fundamental theories and tools. This is the canonical starting point for Isabelle applications.
3. `HOL-Analysis`: classical analysis based on topology etc. (see http://isabelle.in.tum.de/dist/library/HOL/HOL-Analysis).
4. `HOL-Probability`: probability theory based on integration and measure theory etc. (see http://isabelle.in.tum.de/dist/library/HOL/HOL-Probability).

The internal theory structure of `HOL-Analysis` is visualized in figure 1. Each node is one big theory file: a total of 72 files and 6.5 MB source text. Processing all of it on 8 CPU cores requires approx. 4.5 min elapsed time, 25 min CPU time, and 3.5 GB memory.

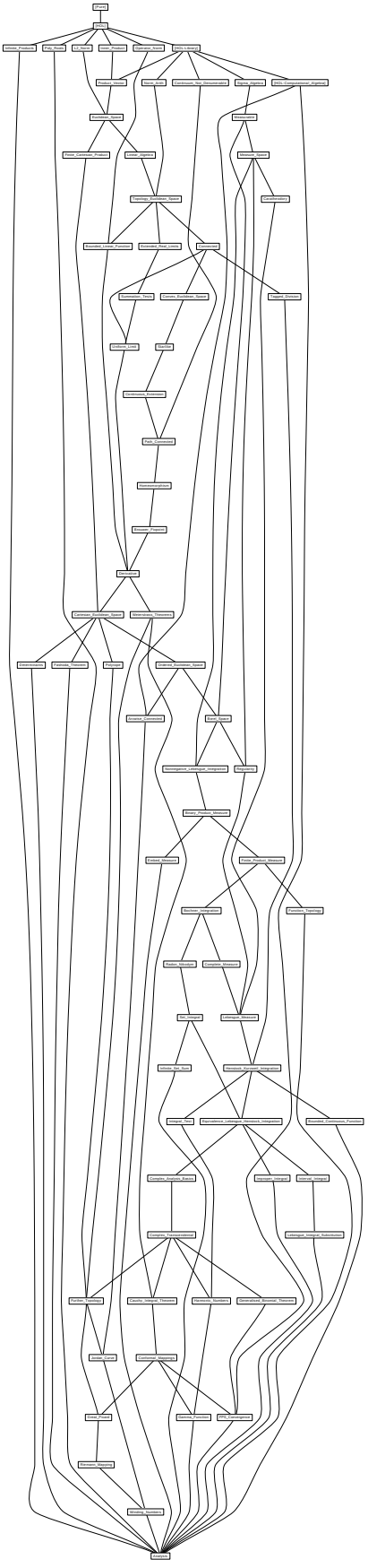

Figure 1: Theory and session imports in `HOL-Analysis`

1.2 The Archive of Formal Proofs (AFP)

The *Archive of Formal Proofs* (AFP) https://www.isa-afp.org collects Isabelle sessions produced by users: it is organized like a scientific journal. At the same time, it serves as library of formalized mathematics: contributors are welcome to build on existing work. The AFP has grown substantially in recent years, now approaching approx. 400 articles (sessions) and approx. 300 authors, see also https://www.isa-afp.org/statistics.html.

Figure 2 on page 13 shows all sessions of Isabelle + AFP that are connected to `HOL-Analysis` wrt. the parent-session relation: a total of 75.

Figure 3 restricts this graph to sessions connected to `Ordinary_Differential_Equations` (and its clone `HOL-ODE`), which extends `HOL-Analysis` by theories for ordinary differential equations, including verified algorithms for numeric approximations. This development ultimately leads to sessions `Lorenz_C0` and `Lorenz_C1`, which perform symbolic reasoning by computation and require approx. 50 h CPU time total.

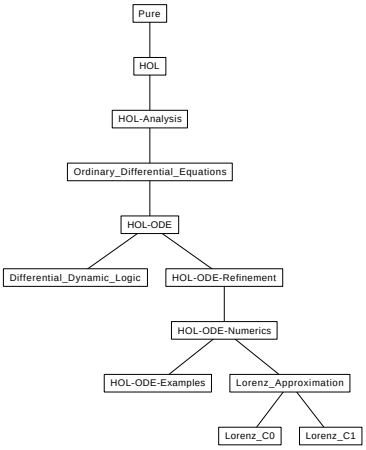

Figure 3: Sessions connected to `Ordinary_Differential_Equations`

When developing such applications, the path through the parent-session hierarchy is critical. Figure 4 shows resource requirements for the full stack leading up to session `Lorenz_Approximation` (8 CPU cores, 64 bit Poly/ML with several GB memory). All of this needs to be processed on the spot, when loading e.g. `Lorenz_C0` into the Prover IDE for the first time.

session	stored heap	elapsed time	CPU time
`Pure`	17 MB	0.2 min	0.2 min
`HOL`	227 MB	2.7 min	9.3 min
`HOL-Analysis`	127 MB	4.4 min	24.5 min
`Ordinary_Differential_Equations`	54 MB	2.0 min	8.3 min
`HOL-ODE`	3 MB	0.0 min	0.0 min
`HOL-ODE-Refinement`	151 MB	3.8 min	21.4 min
`HOL-ODE-Numerics`	110 MB	15.8 min	35.8 min
`Lorenz_Approximation`	22 MB	3.5 min	7.6 min
total	711 MB	32.4 min	107.1 min

Figure 4: Cumulative resource requirements for `Lorenz_Approximation`

If we now envisage a formalization of Financial Mathematics (e.g. Itô calculus) it might start with existing `HOL-Analysis` or `HOL-Probability` and develop add-ons similar to the `HOL-ODE` hierarchy — and much more. It is clear that more scaling of the technology is required to support such applications conveniently.

2 Common modes of operation

The development of a library of formal proof documents, which is based on contributions by other people and subject to continuous changes, involves the following modes of operation:

Editing of many theories from different sessions, with immediate feedback from the prover process.

Building of session images and databases in batch-mode.

Browsing existing library content.

Isabelle provides substantial support for *editing* and *building*, namely the Prover IDE (§2.1) and tools for batch-builds (§2.2). Both have slightly different profiles of resource usage, and different demands for further scaling.

In contrast, *browsing* (§2.3) is somewhat underdeveloped: HTML and PDF documents are generated in batch-builds, but the content only refers to superficial syntax. Full semantic annotations could be "browsed" in the Prover IDE, but this requires costly reprocessing in interaction, Proper PIDE browsing could be based on a HTTP server for the document model with its markup.

2.1 Editing: Prover IDE

After download of the main Isabelle application, users first encounter the Isabelle/jEdit front-end (see figure 5, and figure 6 on page 14). This semantic editor is based on an agglomerate of technologies called Isabelle/PIDE. See also the Isabelle/jEdit manual [5]; PIDE concepts are further explained in [6] and [7].

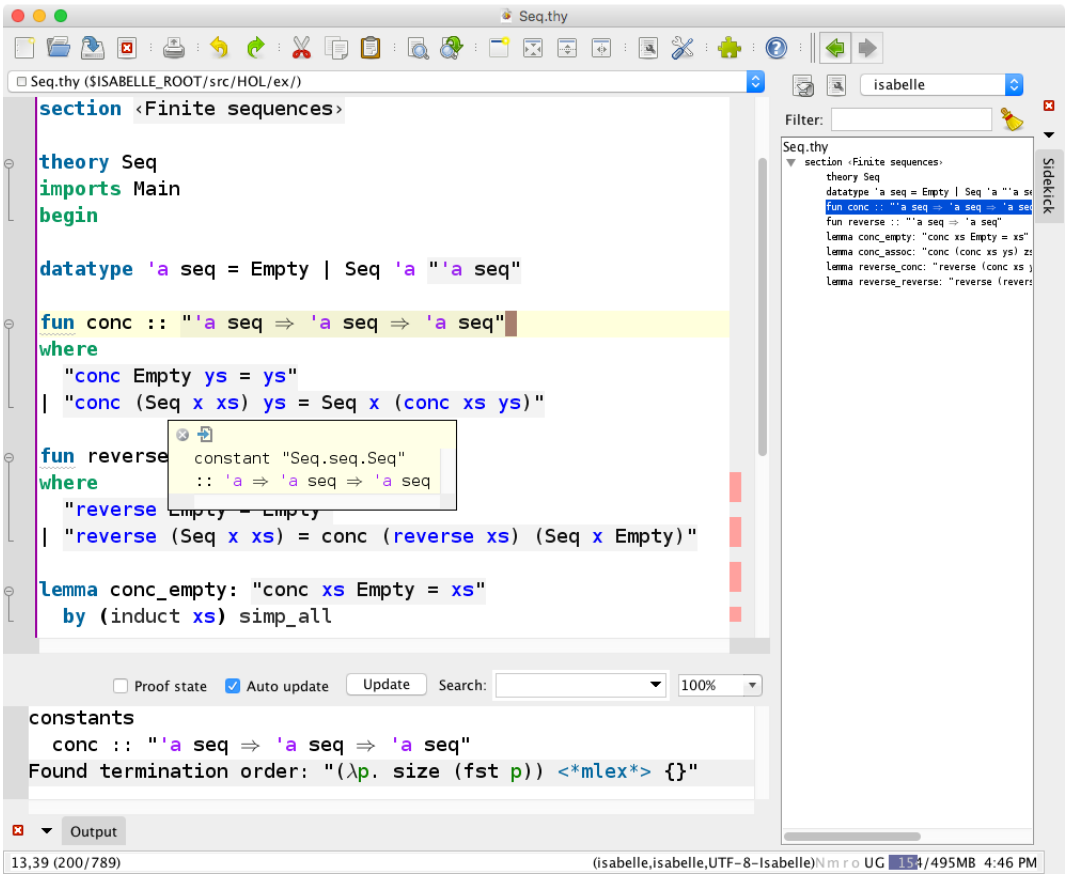

Figure 5: The Isabelle/jEdit Prover IDE

Isabelle/PIDE provides an impression of direct editing of formal document content, while the prover is continuously checking in the background. This resembles an advanced "spell-checker" for documents of formalized mathematics, or any other language that is embedded into Isabelle theories. There is even a conventional spell-checker for comments written in English.

The Prover IDE operates on whole projects (sessions), which may consist of hundreds of theory files, with a typical size of 50–500 KB for each theory.

Isabelle users can get started with a solid consumer laptop with 4 CPU cores and 8 GB memory, but this is barely sufficient for medium-sized sessions like `HOL-Analysis`. For resource requirements and scalability of interactive PIDE sessions, the following main factors are relevant:

The Isabelle/ML process for the prover back-end. It runs in 32 bit mode by default, even though a proper 64 bit platform is now required for the Isabelle application. Thus ML can access a total of approx. 3.5 GB stack + heap space: the process starts with 500 MB and expands or shrinks the address space according to the current demands; this is a consequence of normal memory management and garbage collection. When the heap becomes critically full, memory is reclaimed by *sharing substructures* of immutable ML values: this is possible thanks to the clean mathematical semantics of ML. Situations of heavy-duty memory management can cause notable pauses during interaction, usually in the range seconds up to half a minute.

For very big applications, the 32 bit model is no longer feasible. Running the ML process in 64 bit mode requires almost double as much memory, due to uniform representation of values and pointers as one machine word. So 64 bit ML only makes sense for hardware with at least 16–32 GB memory.

The Isabelle/Scala process for the Prover IDE front-end. The jEdit editor runs on the same Java Virtual Machine (JVM). The PIDE markup for theory content that is produced by the prover is stored in Isabelle/Scala as one big markup tree. Whenever the editor renders text or reacts to mouse events, it needs to retrieve that information in real-time (1–10 ms). This works well up to a certain session size, but when the heap of the JVM fills up, it can become unresponsive or even unstable.

By default, the heap size for the Isabelle/jEdit application is restricted to 0.5–2.5 GB. This is a concession to average users with average hardware, in order to get started without manual configuration. Increasing the JVM heap boundary requires a restart of the application, see also chapter 7 "Known problems and workarounds" in the Isabelle/jEdit manual [5]. For medium-sized sessions like `HOL-Analysis` the default JVM heap sizes should be doubled.

Much of the potential for scaling is based on careful inspection how the Isabelle/ML and Isabelle/Scala processes really work, both by adjusting tuning parameters and by more profound reforms, see also §3.

2.2 Building: batch-mode

Batch builds are implicit in the Prover IDE: after startup of Isabelle/jEdit there is a check if the specified session heap file is up-to-date wrt. its sources. Otherwise, it is built on the spot, notably on the first start of the Isabelle application. This can take approx. 3 min for the default `HOL` session. Users can also switch Isabelle/jEdit to a different base session, e.g. `HOL-Analysis` or `HOL-Probability`. This requires a restart of the application — or a slightly tricky reload of the Isabelle plugin within jEdit — in order to trigger the batch-build process again and load the specified session image.

The Isabelle command-line tool `isabelle build` provides numerous options to manage batch-builds; the tool is documented in chapter 2 of the Isabelle System Manual [3]. That is particularly relevant for building many sessions simultaneously, using multiple build processes (option `-j`*N*) each with multiple threads (option `-o threads=`*M*). On high-end multicore hardware (say with 24 cores), this allows to build the full Archive of Formal Proofs in less than 1 h elapsed time (excluding the specifically marked group of `slow` sessions).

Advanced applications of `isabelle build` should avoid fancy shell scripting, but use the underlying Isabelle/Scala functions directly: `Build.build()`, `Sessions.load()`, `Sessions.deps()`. An example for such Isabelle/Scala system-programming is the infrastructure for nightly builds of Isabelle + AFP: results are presented at http://isabelle.in.tum.de/devel/build_status. The implementation uses simple and convenient Isabelle/Scala modules for SSH remote connections, Mercurial source code management, and SQL database access (SQLite and PostgreSQL).

Batch-builds are traditionally presented as a closed process, to produce the required heap images for a PIDE session, or to test many sessions from a library. From the perspective of scalability, it would be convenient to allow *transitions* between interaction and batch-builds in two directions:

1. A failed batch-build could be turned directly into an editing session, without starting it again, see also §3.9.
2. A finished IDE session could be saved as a heap image, for re-use in other IDE sessions or batch-builds, see also §3.10.

This requires more substantial changes of how Isabelle manages session images, but the underlying Poly/ML system already supports such mixed modes of operation.

2.3 Browsing: client-server applications

In contrast to editing and building, browsing may be characterized as *read-only* access to *existing* content of the library, usually with a more *light-weight front-end* than a full-scale IDE, and potentially with *better rendering quality* than plain text. Current HTML browsers have the potential to deliver this, but Isabelle only provides rather old-fashioned static HTML that visually resembles the syntax highlighting in Isabelle/jEdit, e.g. see http://isabelle.in.tum.de/dist/library/HOL/HOL-Analysis/Sigma_Algebra.html.

Isabelle/jEdit already provides an action `isabelle.preview` that does similar HTML rendering of the current theory buffer. It uses semantic markup of PIDE and thus provides more details of nested languages. This is implemented via an HTTP server within the Prover IDE: the preview command opens a standard web-browser on a URL that points to the internal document model.

An alternative is the experimental Isabelle/VSCode front-end for Isabelle/PIDE, which has been published with Isabelle2017 (October 2017) for the first time (see also https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2017). Visual Studio Code is an open-source project by Microsoft; it is based on the Electron platform, which consists of the Chromium web-browser with Node.js runtime system. The resulting application is a plain-text editor with some extra styles and text markup, but it is also possible to produce HTML5 previews on the spot, see figure 7 on page 15 (again with the same old-fashioned HTML output of Isabelle).

This means, VSCode is an editor and a browser at the same time. The rendering quality in Isabelle/VSCode is still below Isabelle/jEdit, but the underlying Chromium platform has the potential to approach the typesetting quality of mathematical textbooks or journals, together with semantic markup and hyperlinks as usual for websites. VSCode is also notable for its Language Server Protocol, which is based on JSON-RPC and publicly maintained by Microsoft. Thus the PIDE document model becomes accessible by a public protocol, outside of the Isabelle/Scala programming environment.

Extrapolating current possibilities for browsing a bit further leads to interesting application scenarios:

1. Remote HTTP service for Isabelle/PIDE, with regular web-browser as local client: strictly for browsing HTML + CSS + JavaScript, but no editing. The server retrieves PIDE markup for theories from a database that has been produced by batch-builds beforehand. The server does not require a prover process. The client does not require an editor.
2. Remote Isabelle/PIDE service with a custom display protocol — similar to the Language Server Protocol of VSCode — with various local editor front-ends:
   (a) Local Isabelle/jEdit without Isabelle/ML and without the full Isabelle/Scala markup tree. The user merely has a medium-sized JVM application (2 CPU cores, 2 GB memory) with semantic markup restricted to the open theory buffers in the editor. The Isabelle/ML prover process and the Isabelle/Scala PIDE process with full markup information run on the server (several CPU cores and several GB memory).
   (b) Local Isabelle/VSCode with minimal Isabelle/Scala process to connect to the PIDE server as above, with similar resource requirements. Here the local application still consists of two runtime environments: VSCode on Node.js / JavaScript and Isabelle/Scala on the Java VM.

(c) Local Isabelle/VSCode without the Isabelle/Scala JVM process. Everything runs within the Node.js environment of VSCode. The required Scala modules for communication with the Isabelle/PIDE server are translated to JavaScript using ScalaJs https://www.scala-js.org — but note that this is relatively new emerging technology. This approach has the potential to reduce local resource requirements by 50% (1 CPU core, 1 GB memory) and require less disk space by omitting the local JVM installation.

The web-client from point 1 above could in principle be generalized towards an editor (or IDE) that runs within common web-browsers, but I consider this merely a theoretical possibility due to the "HTML browser hell". There are too many different browsers in different versions, and diverging interpretations of various web-standards. Projects for web-based IDEs exist, but are still lagging behind "real" desktop applications.

In contrast, point 2(c) has better prospects to achieve a browser-based IDE: there is only one Chromium engine in VSCode, and the whole application may be packaged for users to deliver exactly one well-defined version. Thus it becomes a web-application that is delivered like a traditional desktop application, where everything is properly integrated and tested. Microsoft distributes VSCode under the slogan: "Code editing. Redefined. Free. Open Source. Runs everywhere.", which raises the expectation that it should work smoothly on all platforms, like standard Firefox or Chromium browsers.

3 Technical approaches to scaling

The subsequent approaches to scaling take technological side-conditions as a starting point, and sketch possibilities to improve certain aspects of the overall system. This is a bottom-up view on the problem.

3.1 Skipped proofs in the Prover IDE

Status: ***realistic, short-term***

Approach: Important repository versions are fully processed in the background, e.g. "nightly builds". Successful results are recorded by a central database server (PostgreSQL). A local repository pull can be re-checked quickly by omitting proofs that have already been checked before: implicit **sorry** commands are inserted by the system, according to database content that is retrieved on the spot. This refines the existing system option `skip_proofs`, which either skips *all* proofs or *none*.

Consequences: Better IDE performance when editing existing library sessions: only the relevant non-proof parts need to be checked, in order to explore the formal context in a particular situation. The timings in figure 4 (all proofs checked) vs. figure 8 (all proofs skipped) provide some hints about the potential of this approach. Sessions like `HOL-Analysis` with many conventional proofs benefit greatly, but other sessions like `HOL-ODE-Numerics` consist of heavy computations or synthesis of results that need to run in full.

session	stored heap	elapsed time	CPU time
`Pure`	17 MB	0.2 min	0.2 min
`HOL`	209 MB	2.2 min	3.9 min
`HOL-Analysis`	103 MB	1.3 min	3.2 min
`Ordinary_Differential_Equations`	47 MB	1.0 min	1.9 min
`HOL-ODE`	3 MB	0.0 min	0.0 min
`HOL-ODE-Refinement`	138 MB	1.9 min	6.7 min
`HOL-ODE-Numerics`	104 MB	13.5 min	16.5 min
`Lorenz_Approximation`	19 MB	2.8 min	3.7 min
total	640 MB	22.9 min	36.1 min

Figure 8: Cumulative resource requirements for `Lorenz_Approximation` with `skip_proofs`

Conclusion: A straight-forward continuation of the original "trusted inference kernel" approach of LCF,

extended to a trusted database of already finished proofs. Great potential to speed up the development cycle. Final checking of results can be still done without in batch-mode, without such shortcuts.

3.2 Forked proofs in the Prover IDE

Status: ***realistic, mid-term***

Approach: The full model of parallel proofs (and sub-proofs) of Isabelle batch-builds is integrated into the interactive Prover IDE. So far the parallelism of PIDE has been limited to theory graph structure, terminal proof steps (**by** *method*), and asynchronous print functions (e.g. implicit *print_state* or *sledgehammer* via GUI panel).

Consequences: Faster proof processing in the Prover IDE on machines with many cores. Less impact on under-powered machines.

Conclusion: This old idea has not been implemented yet, because most users have under-powered machines and require performance tuning in different areas.

3.3 Swapping of PIDE markup database: JVM heap vs. external database

Status: ***realistic, short-term***

Approach: Instead of accumulating semantic PIDE markup within Isabelle/Scala (which is bounded by the JVM heap) there is an external database to absorb theories that are presently unused in the editor. In effect, JVM data is swapped in and out wrt. a database file (SQLite). A more ambitious approach could also use a full database server (PostgreSQL).

Consequences: Reduced load on the JVM, which does not scale beyond a few GB. The conceptual model to apply PIDE to a whole library like AFP has better chances to succeed.

Conclusion: A worth-while continuation of recent efforts to unify the data model of batch-builds with the Prover IDE, with solid underpinning by well-established database technology that is already integrated into Isabelle/Scala.

3.4 Distributed batch-builds

Status: ***realistic, short-term***

Approach: The idea is to connect *equivalent* Isabelle installations via `ssh` command-line execution: a local user runs a local Isabelle/Scala tool that invokes `isabelle build` remotely and downloads the resulting heap image — which could then be used in a local Prover IDE session.

Consequences: Much faster building of an IDE context, if the local machine is relatively slow, the remote server is very fast, and the network connection fast enough for 100–1000 MB download of heap images.

Conclusion: This approach is simple and useful. It is already used by the developers of IsaFoR/CeTa http://cl-informatik.uibk.ac.at/software/ceta as an exercise in Isabelle/Scala system-programming, combined with the `rsync` tool. It is not yet part of the official Isabelle release, because the notion of "equivalent" Isabelle installations on the remote system requires further clarification.

3.5 Heap hierarchy compression: skipping intermediate sessions

Status: ***emerging, short-term***

Approach: Thanks to session-qualified theory names in Isabelle2017, the theory name space is now independent of the accidental session heap hierarchy. This allows the Isabelle build process to rearrange the composition of sessions on the spot. Shortly after the Isabelle2017 release, options such as `isabelle`

`jedit -A` *ancestor_session* `-S` *target_session* have emerged, in order to specify an interval in the session stack that should be turned into just one heap image. Further refinements of this idea are possible, especially a combination with distributed builds (§3.4).

Consequences: Faster local builds where session heap hierarchies are overly complex or contain many theories that are not required in the final image.

Conclusion: Very useful reform, based on simple reorganization of the local build process, only small changes of existing Isabelle system tools.

3.6 Headless PIDE server for external tool integration

Status: ***emerging, short-term***

Approach: The main API operations for PIDE edits and batch-builds are unified within a headless server process that understands JSON-RPC, for example.

Consequences: External tools can easily communicate with a continuously running Isabelle process. Costly startup of toplevel command-line tools is avoided.

Conclusion: Relevant moves towards better integration of Isabelle into other tool environments.

3.7 Isabelle Docker container

Status: ***emerging, short-term***

Approach: If we want to move Isabelle into the cloud, it could be helpful to use a standard container format. Isabelle2017 (October 2017) already provides an official Docker image https://hub.docker.com/r/makarius/isabelle, which contains a regular Ubuntu Linux installation and Isabelle2017 with HOL image.

Consequences: Instead of ensuring manually that an Isabelle installation on Linux sees the required shared libraries for C/C++ in 32 bit mode, and a few standard tools like `curl` and `perl`, it is possible to work with explicitly defined system images. This is sometimes convenient or even required for cloud services, but it demands much more file-system space than conventional installation on an existing Linux host. Docker containers also provide a checkpoint facility, but it requires much more disk space than "saved world" images of the underlying Poly/ML system.

Conclusion: Use it when really required. Otherwise work more efficiently with direct installation on the host operating system, together with Isabelle session images and database files as required.

3.8 Fork of ML processes instead of heap load/store/load cycles

Status: ***speculative, mid-term***

Approach: Instead of restarting ML build processes from scratch, with full load/store/load cycles of the heap image, the build tree is represented by a process tree of Unix forks (on Linux or Mac OS X).

Consequences: Potential performance improvement of large build jobs with many sessions. Heap images become strictly optional. The true performance impact remains to be seen, as Poly/ML 5.7.1 (November 2017) already improves the load time of heap images significantly.

Conclusion: Something left in the back-hand, when the pressure on AFP builds increases again beyond a tolerable amount of time.

3.9 Direct transition from failed batch-build to IDE session

Status: ***realistic, mid-term***

Approach: The partial result of a failed batch-build is not thrown away, but turned into a heap image that can be used in a PIDE session, to let the user address problems encountered in the previous attempt.

Consequences: Faster maintenance cycle for big libraries. Local or remote batch-build jobs can provide the context for PIDE sessions more directly, than reloading from scratch.

Conclusion: Beneficial reform around the existing dumped-world model of Poly/ML, where build artefacts are not really meant to be static, but a dynamic database of ongoing work.

3.10 Direct transition from finished IDE session to saved heap image

Status: ***realistic, mid-term***

Approach: The PIDE session is made persistent as a heap image. This requires careful shutdown of the parallel PIDE protocol handler in ML, and storing of intermediate results persistently within the ML world, similar to the classic batch-mode theory loader.

Consequences: Both interactive PIDE sessions and batch-build jobs may reuse interactively produced session content.

Conclusion: Beneficial reforms around the existing dumped-world model of Poly/ML, where build artefacts are not really meant to be static, but a dynamic database of ongoing work.

3.11 Upgrade of PIDE edits and markup to Mercurial changesets with external database

Status: ***realistic, mid-term***

Approach: The existing PIDE document model — with its edits and markup in Isabelle/Scala — is upgraded to work with Mercurial changesets and markup stored in external databases (SQLite or PostgreSQL).

Consequences: Semantic annotations for the persistent history of sources. Support for multi-version editing within the same Prover IDE session.

Conclusion: A natural extension of PIDE concepts that open the perspective towards distributed editing and browsing via distributed version control.

3.12 Lazy Isabelle context elements (locales)

Status: ***realistic, short-term***

Approach: Some applications use a lot of internal context structure, via locales and locale interpretation: it can require minutes just to revisit a complex context hierarchy, in order to prove new results. This could be made more efficient by using *lazy context elements* for the main conclusions (**notes** within a context).

Consequences: Only strictly required locale facts are produced on demand. Big and complex sessions like `JinjaThreads` in AFP could become much faster (presently approx. 1 h elapsed time and 3 h CPU time on 8 cores).

Conclusion: Beneficial reduction of resource requirements, which requires modest reforms of the internal management of local theory contexts in Isabelle/Pure.

3.13 Distributed Poly/ML processes with virtual shared memory

Status: ***speculative, long-term***

Approach: The Poly/ML runtime system is upgraded to distributed processes on multiple CPU node, e.g. on a compute cluster with distributed memory and explicit message passing. The impression of shared memory is reconstructed from implicit copying of data and demand. This is feasible, because the majority of ML data structures are immutable and subject to mathematical equality of content, not physical location.

Consequences: Even more scaling of multi-threaded ML programming, beyond a single "fat node" towards genuine cluster computing. The cost for that is considerable complexity added to ML memory management at run-time.

Conclusion: The classic paper on parallel Poly/ML and Isabelle/ML [1] briefly discusses possibilities of distributed and parallel memory management, based on much earlier experiments [2]. Revisiting all this today is probably a much bigger effort than anticipated in the past, when the underlying system was more primitive.

3.14 Reducing Poly/ML memory footprint on native 64 bit

Status: ***realistic, mid-term***

Approach: The idea is to use native `x86_64` machine instructions in ML, but refer to heap objects via 32 bit indexing instead of native addresses. Together with some bit shifting, this would allow to use 16 GB heap space in terms of the classic 32 bit model (which is presently limited to 3.5 GB). See also the JVM approach to "Compressed object pointers" https://wiki.openjdk.java.net/display/HotSpot/CompressedOops.

Consequences: Scaling-up the available heap space without penalty for full 64 bit values. Avoid dependence on legacy 32 bit C/C++ libraries, as Linux, Windows, mac OS are slowly phasing out `x86` compatibility.

Conclusion: David Matthews (architect and maintainer of Poly/ML) has already made initial experiments to show that the approach works, but substantial efforts would be required to push it all through the Poly/ML compiler and run-time system. Feasibility also depends on the funding for such a project.

References

[1] 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)*. http://www4.in.tum.de/~wenzelm/papers/parallel-ml.pdf.

[2] David C. J. Matthews & Thierry Le Sergent (1995): *LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection*. In: *Memory Management, International Workshop IWMM 1995, Kinross, UK*, pp. 297–311. https://doi.org/10.1007/3-540-60368-9_30.

[3] Makarius Wenzel: *The Isabelle System Manual*. http://isabelle.in.tum.de/doc/system.pdf.

[4] Makarius Wenzel: *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/doc/isar-ref.pdf.

[5] Makarius Wenzel: *Isabelle/jEdit*. http://isabelle.in.tum.de/doc/jedit.pdf.

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

[7] Makarius Wenzel (2014): *System description: Isabelle/jEdit in 2014*. In Christoph Benzmüller & Bruno Woltzenlogel Paleo, editors: *User Interfaces for Theorem Provers (UITP 2014)*, EPTCS. http://eptcs.web.cse.unsw.edu.au/paper.cgi?UITP2014:11.

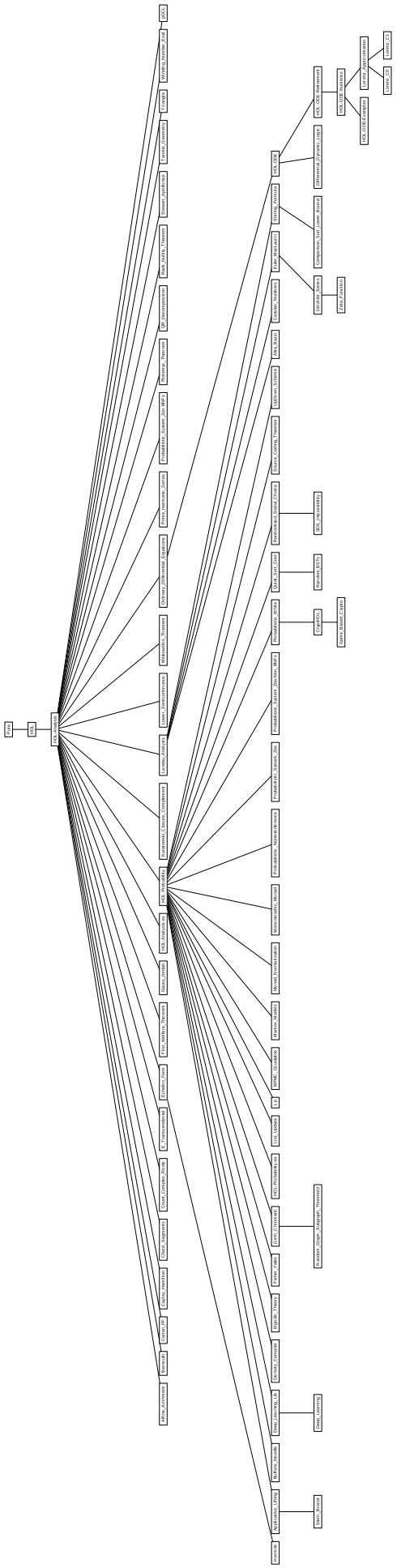

Figure 2: HOL-Analysis session graph (wrt. parent-session relation)

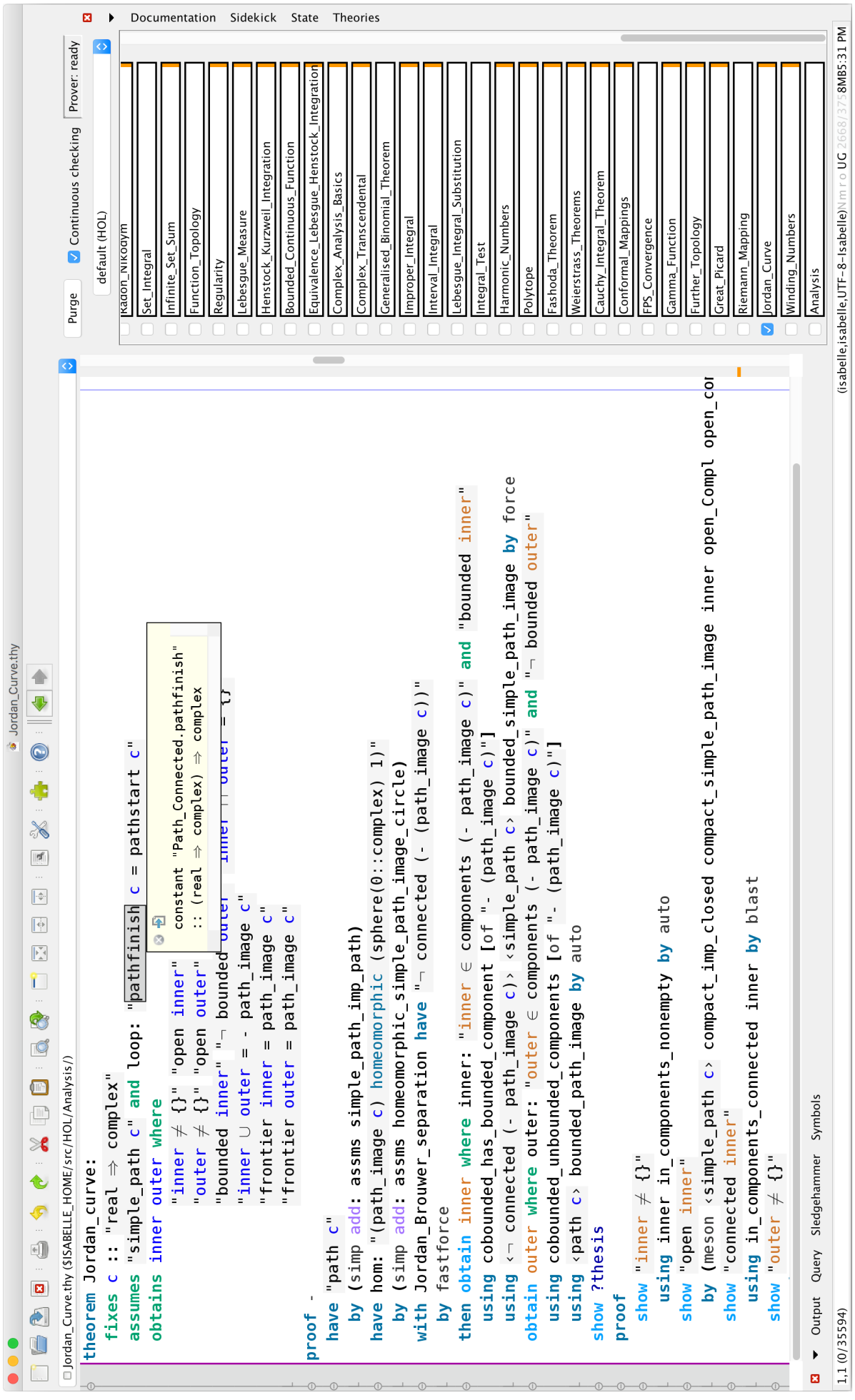

Figure 6: Session `HOL-Analysis` within Isabelle/jEdit (ML process: 2.8 GB, JVM process: 3.5 GB)

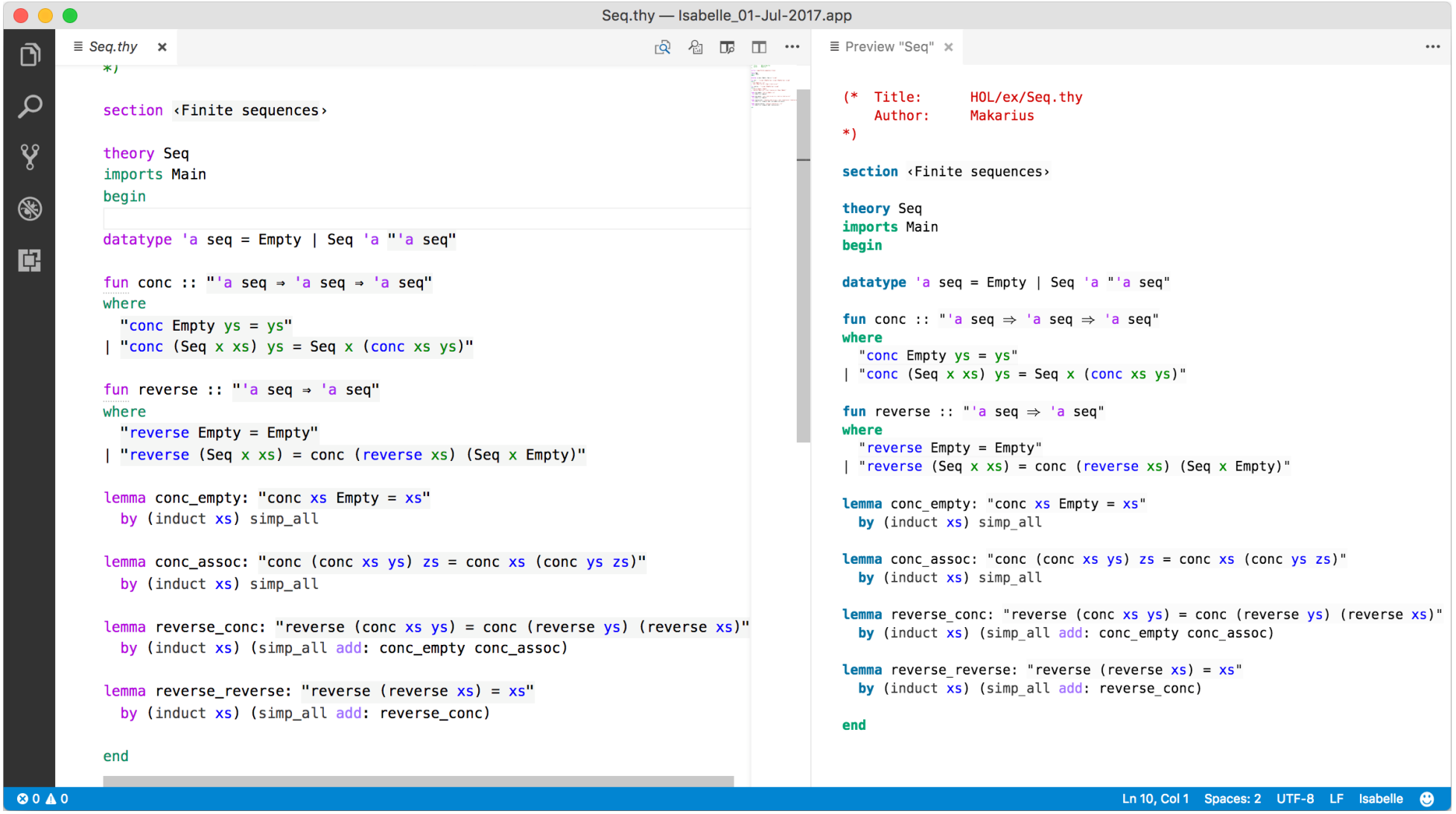

Figure 7: Isabelle/VSCode Prover IDE with built-in HTML preview (Chromium)