Further Scaling of Isabelle Technology

Makarius Wenzel
April 2018

http://sketis.net

Abstract

Over 32 years, Isabelle has made a long way from a small experimental proof assistant to a versatile platform for proof document development. There has always been a challenge to keep up with the natural growth of applications, notably the Archive of Formal Proofs (AFP). Can we scale this technology further, towards really big libraries of formalized mathematics? Can the underlying Scala/JVM and Poly/ML platforms cope with the demands? Can we eventually do 10 times more and better? In this paper I will revisit these questions particularly from the perspective of:

- Editing: Prover IDE infrastructure and front-ends,
- Building: batch-mode tools and background services,
- Browsing: HTML views and client-server applications.

1 Introduction

(The underlying Isabelle version for this paper is f69ea1a88c1a from the repository[1], as approximation of the Isabelle2018 release (probably August 2018).)

1.1 Isabelle as framework of domain-specific formal languages

Isabelle is usually presented as a *proof assistant*, while historically important papers [3, 4] classify the system as *Generic Theorem Prover* or *Logical Framework*. In the past 10 years, I have usually understood the "framework" aspect in a technological sense, as a platform for *domain-specific formal languages*. The logical language is just one (sophisticated) example for that.

There is a general tendency towards self-application: Isabelle is a platform to build Isabelle libraries and tools, or to write books and papers about Isabelle. The Prover IDE (PIDE) has become increasingly important for that, although it is only implicitly present in the subsequent categories of domain-specific languages for Logic, Programming, and Proof.

[1] https://isabelle.in.tum.de/repos/isabelle

Category: Logic

Isabelle/Pure is the logical framework and bootstrap environment. The Pure logic starts out as Minimal Higher-Order Logic, which is used to represent rules for of Higher-order Natural Deduction declaratively. Rule composition works by backchaining and higher-order unification.

Isabelle/HOL is the main library of theories and tools for applications. The initial axiomatic basis resembles the original HOL by Gordon [1], but it is built as a library within Isabelle/Pure and inherits from its distinctive style. Consequently, Isabelle/HOL is quite different from the HOL family of provers (HOL4, HOL-Light, etc.). End-users need to care little about logical foundations of Isabelle/HOL: they see high-end tools for specifications and proofs (e.g. Sledgehammer).

Category: Programming

Isabelle/ML is the Isabelle tool implementation and extension language. It is based on Poly/ML[2] by David Matthews, but heavily augmented to carry the weight of the Isabelle framework. Both Isabelle/Pure and Isabelle/ML emerge from the same bootstrap process: the result is a meta-language for programming the logic that is intertwined with it from a technological viewpoint, but logic and programming remain formally separated.

Isabelle/Scala is the Isabelle system programming language. Scala[3] is a functional-object-oriented language that runs on the Java Virtual Machine. Isabelle/Scala connects the logical environment with the outside world of TCP servers, database engines, GUI environments etc. while retaining the style and manner of Isabelle/ML.

Category: Proof

Isabelle/Isar is the structured proof language of the Isabelle framework — Isar means Intelligible semi-automated reasoning. The initial language design from 1998–2001 has been renovated and extended in 2015–2016[4] and augmented by the Eisbach language for proof methods [2].

Document language for HTML output and LaTeX type-setting of proof text. Nice documents are the main result of Isabelle proof production. A *proof document* documents what has been proven to the general audience, both in formal and informal text.

[2] http://polyml.org
[3] https://www.scala-lang.org
[4] https://sketis.net/2016/the-isar-proof-language-in-2016

1.2 Isabelle applications: The Archive of Formal Proofs

The Archive of Formal Proofs (AFP)[5] is the main repository of Isabelle applications: it is organized as a Scientific Journal, and may be seen as a library of formalized mathematics for Isabelle. AFP was founded in March 2004 and has grown to considerable size: the amount of material (and build times) have approximately doubled in the past 3 years. Here are some statistics from 26-Mar-2018 (afp-devel version e9aaf72d221c)[6]:

- 275 authors
- 409 articles
- 4208 theories
- 10^5 theorems
- 10^8 bytes of text
- build time (without group `slow`):
  - 22 h CPU time
  - 65 min elapsed time on 22 cores (factor 20)
- build time (group `slow` without `very_slow`):
  - 15 h CPU time
  - 95 min elapsed time on 22 cores (factor 9.5)

AFP entries are updated in correspondence to ongoing Isabelle development. For example, a minor change in Isabelle command syntax, logical notation, or rule declarations eventually requires a full test on AFP. This also serves as a reality check of tentative changes to Isabelle and its libraries: if AFP applications turn out too difficult to upgrade, changes are be revised or rejected.

Good response times on such tests are vital for further development and maintenance of Isabelle and AFP. As a rule of thumb the following practical *time scales* have emerged:

Max. 45 min: online time for quasi-interactive builds while sitting at the computer and doing other things. This time span is also anecdotal as the "Paris commuter's constant", i.e. the practical limit of a commuter sitting patiently on a train to wait for its arrival.

Max. 120 min: offline time for batch-builds while being absent and not watching it. This time span coincides with the classic French lunch break.

According to the build times for AFP above, we are again at the limits of comfort (on high-end hardware). The problem could be solved by throwing more cores at it, investing in even better hardware. Another approach is to look systematically for possibilities of

[5] https://www.isa-afp.org
[6] See also https://devel.isa-afp.org/statistics.html

scaling the underlying technology. Probably both needs to be done at some point, with regard to continued growth and prosperity of AFP.

2 Technologies contributing to Isabelle

Technology can be both beneficial and dangerous: it can become a burden for continued development and maintenance. Great care is required to select adequate technologies and to turn them into proper use. By default, fancy new things with a lot of publicity around it should be questioned thoroughly.

Here follows an overview of notable technologies that contribute to Isabelle in different states of adoption: *established*, *emerging*, *experimental*. The more Isabelle depends on any of these, the more resources need to be invested to keep them alive and prospering.

2.1 Established technologies

- Isabelle/ML: based on Poly/ML (by David Matthews)
- Isabelle/Scala: based on the Java VM (by Oracle)
- Isabelle/jEdit: based on the jEdit text editor (in Java)
- Mercurial: source control management (SCM)

Isabelle/ML is a distinctive member of the ML family, with a rich library and high-end IDE. Since April 2016, the IDE can load the Isabelle/ML/Pure bootstrap environment into itself, which greatly simplifies further development.

At the bottom of Isabelle/ML is Poly/ML by David Matthews. Started in 1985 as one of the first implementations of Standard ML, Poly/ML and has gone through many phases of improvements and further scaling, often specifically for Isabelle. After decades of performance tuning, it is now hard to imagine a different ML platform to carry the weight of Isabelle applications: it would mean a loss of one or two orders of magnitude in performance.

Isabelle/Scala is a library for Isabelle system programming based on regular Scala, which is hosted on the Java platform (maintained by Oracle). Isabelle/Scala continues the functional programming style of Isabelle/ML on the JVM platform. There is an overlap of many fundamental modules with Isabelle/ML (e.g. to manage files and processes), but the main purpose of Isabelle/Scala is to connect to the outside world (servers, databases, GUIs etc.).

Although the main body of Isabelle/PIDE is implemented in Isabelle/Scala, its own development model is rather old-fashioned, using a plain text-editor and command-line invocation of batch-builds.

In recent years, Isabelle/Scala has also become the implementation platform for most Isabelle command-line tools: old-fashioned shell scripts have become very rare.

Isabelle/jEdit is the default front-end for the Isabelle/PIDE framework. The jEdit text editor[7] was once legendary, but its founder left the project in 2006. In the past 5 years, maintenance has continued with approximately 5 remaining enthusiasts: the process still works somehow, but big things cannot be expected any more. My guess is that jEdit can continue in this manner for several years, before users will notice serious dropouts.

Luckily, Isabelle/PIDE is not tied to a particular front-end editor, and experiments towards alternatives have happened with Isabelle/VSCode (see below), but it might require years until this emerging editor platform can compete with Isabelle/jEdit.

Mercurial has become the standard Source Code Management (SCM) system for Isabelle in 2008, when there was still a genuine choice between Mercurial and Git, and not yet the social pressure towards Git seen today.

From the perspective of *technology* (not *sociology*) the reasons for Mercurial are still unchanged: it is more friendly and fits stylistically better to Isabelle. Git might now have more add-on tools, due to a larger community behind it, but Mercurial is still quite active as the second. Basic Mercurial support has already become part of Isabelle/Scala, and more will be required for proper management of theory sources wrt. formal checking by the prover. This is not "compilation" of classic programming languages, but more like an extension of Isabelle/PIDE interaction towards repository management. Thus we need to provide our own setup for this task and cannot just reuse mainstream tools.

2.2 Emerging technologies

- SQL support: Isabelle/Scala with Java Database Connectivity (JDBC)
- SQLite backend: small-scale database files
- PostgreSQL backend: full-scale database server

This is all related to scalable database management, outside of the Poly/ML process (which has its own database format for ML values stored in heap files).

SQL support in Isabelle/Scala goes back to September 2016. The rather concise module `$ISABELLE_HOME/src/Pure/General/sql.scala` provides a wrapper for the standard JDBC interface, to make it look more like Scala than Java. Moreover, the full generality of SQL and JDBC — with its many slightly incompatible backends and different interpretation of data types — is trimmed down to what is really required

[7] http://www.jedit.org

for Isabelle applications. Only two particular backends are supported: SQLite[8] and PostgreSQL[9].

SQLite is a very pupular library for plain-file database management, with sequential access by a single user. It works on many platforms and devices, with supporting libraries for major programming languages and many add-on tools (e.g. as SQLite Browser[10]). Thus SQLite is sometimes positioned as a natural extension of plain-text files, with more structure and scalability. The article "SQLite As An Application File Format"[11] explains why it is better to reuse SQLite to impose some structure on stored data, instead of starting from scratch (or with ZIP as in OpenOffice).

Isabelle/Scala uses the standard JDBC driver for SQLite[12], which is both portable and native on the usual OS platforms: it operates directly on database files within the running Java process.

So far the main Isabelle application is the meta information for session builds, with timing information, hash keys for input sources and output heaps etc. More can be anticipated for the future, e.g. full PIDE markup information stored persistently. Even more ambitious use of SQLite could include the Poly/ML heap database information, although that it is dependent on the OS platform and processor architecture.

PostgreSQL is a high-end database server that supports concurrent access by multiple users, as expected for a proper DBMS. It is very easy to install and maintain as part of the standard Ubuntu server platform (e.g. as appliance on a virtual machine).[13] PostgreSQL is not as well-known than MySQL, but more robust and scalable. Database experts on the Web often recommend PostgreSQL as a conventional database engine with a lot of headroom for scalability, instead of going for newer approaches to "Big Data" via "NoSQL" systems that are less convenient.

Isabelle/Scala uses the standard JDBC driver for PostgreSQL[14], which talks to the database server via a socket connection. It is also possible to forward the connection via the SSH module of Isabelle/Scala, and thus use a remote database in a secure manner, based on regular SSH user authentication.

So far the main application is the database of cumulative Isabelle + AFP build logs since July 2002. Much more can be anticipated, e.g. PIDE markup of processed theory sources that is stored for persistently for various repository versions. Thus the static sources managed by Mercurial SCM are augmented by semantic information stored in the database.

[8] https://www.sqlite.org
[9] https://www.postgresql.org
[10] http://sqlitebrowser.org
[11] https://www.sqlite.org/appfileformat.html
[12] https://bitbucket.org/xerial/sqlite-jdbc/downloads
[13] https://help.ubuntu.com/lts/serverguide/postgresql.html
[14] https://jdbc.postgresql.org

2.3 Experimental technologies

- VSCode: new text editor based on Node.js + Chromium (Electron)
- Scala.js: Scala compiler for JavaScript

Both are related to the JavaScript platform. In 2008 the Isabelle/ML world has been extended towards the Java platform, with the help of Scala. That could be continued by a third leg towards Web technology.

Visual Studio Code (VSCode) is a fast growing open-source project by MicroSoft, with the very liberal MIT license.[15] Despite its name, there is no technological connection to the Visual Studio product: VSCode is a reimplementation in TypeScript on the Node.js + Chromium platform (which is called "Electron"[16] as a spin-off from the Atom editor project).

VSCode aims to redefine the traditional notion of "programmer's text editor" (e.g. as seen in jEdit) towards a new kind of application of semantic editing with "language smartness". Such backends to VSCode may be implemented as regular editor extensions (in TypeScript), or via the new "Language Server Protocol"[17].

These VSCode concepts are very close to Isabelle/PIDE, so it is rather obvious to connect the two. I have already delivered a 1.0 release of Isabelle/VSCode as part of Isabelle2017 (October 2017), although it can hardly compete with Isabelle/jEdit 9.0 in that release.

Scala.js is a version of the Scala compiler that targets JavaScript instead of JVM bytecode. There is also some emulation of basic library modules from the Java API. The current release[18] positions Scala.js rather modestly at version 0.6.x. I have not tried it out myself yet, but got promising reports from some early adopters.

Scala.js opens a perspective to migrate Isabelle/Scala from the JVM towards JavaScript in the future, but right now this is merely speculative. Note that substantial parts of Isabelle/Scala refer to specific system operations from the Java API — this was actually the main point when adding Scala to the portfolio Isabelle technology. Many special tricks in Isabelle/Scala would have to be repeated for the browser or Node.js environment.

As a start, one might try to get some Scala modules for PIDE front-ends onto the JavaScript platform, for regular browser applications or within VSCode.

[15] https://code.visualstudio.com
[16] https://electronjs.org
[17] https://github.com/Microsoft/language-server-protocol
[18] https://www.scala-js.org

3 Further scaling

The quest for further scaling is now turned around: we look at particular application areas of Isabelle, in order to see a potential for improvements:

1. Editing: Prover IDE infrastructure and front-ends,
2. Building: batch-mode tools and background services,
3. Browsing: HTML views and client-server applications.

3.1 Editing: Prover IDE

After download of the main Isabelle application, users first encounter the Isabelle/jEdit front-end (see figure 1, and figure 2 on page 16). Isabelle/jEdit is the main example application of the Isabelle/PIDE framework, and presently the default user-interface.

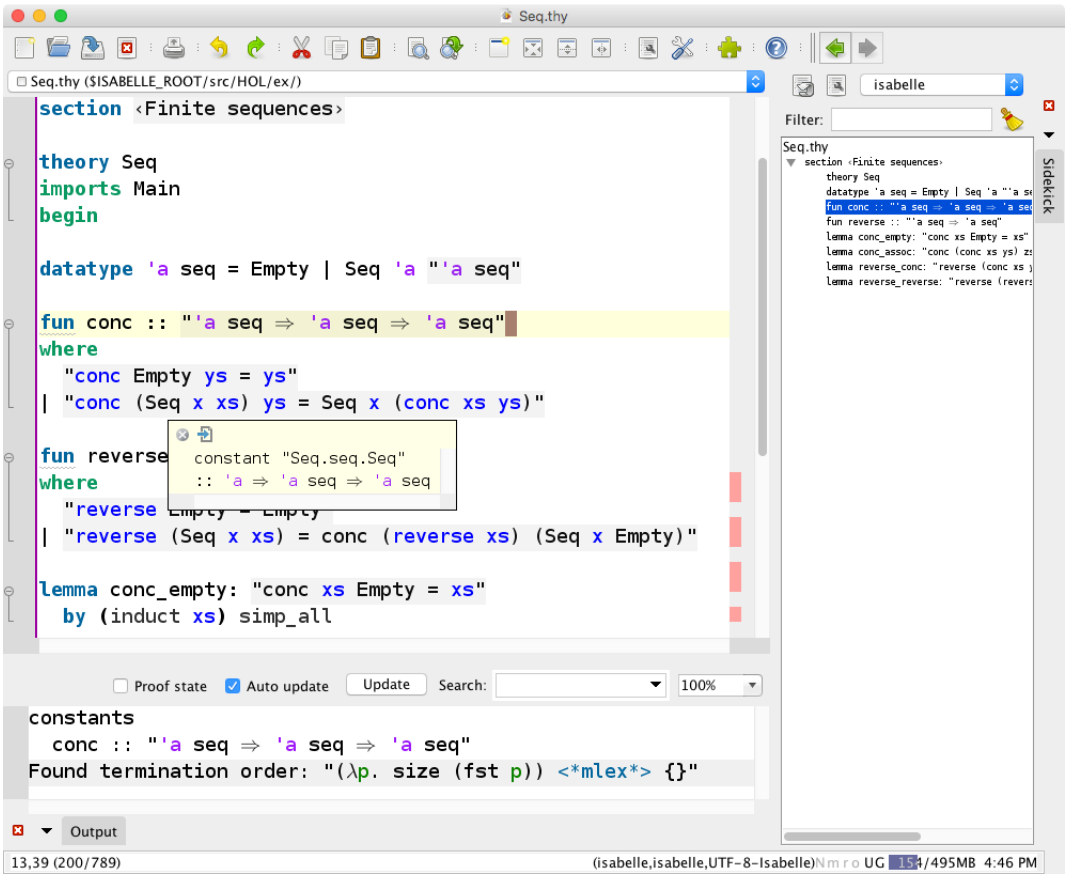

Figure 1: The Isabelle/jEdit Prover IDE

The Prover IDE 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.

Continuous checking within the editor works 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 presently makes only 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 [6]. For medium-sized sessions like `HOL-Analysis` the default JVM heap sizes should be doubled.

Further scaling of the Prover IDE aims at:

- semantic editing of AFP as one big proof document,
- continuous feedback for active sessions,
- markup rendering for passive sessions.

Active sessions are those that may be edited in the current session context, while passive sessions are merely opened as part of the background libraries (e.g. when following a hyperlink to the defining position of a formal entity).

Here are some approaches towards these aims:

- jEdit GUI panels according to session graph structure (Sidekick, Hypersearch, Status with errors / warnings),
- swapping PIDE markup from the JVM heap to an external database (SQLite, PostgreSQL),
- or: internal conversion of PIDE markup (in XML) into XZ-compressed blobs,
- support for skipped proofs and forked proofs: more parallelism,
- PIDE markup for Mercurial changesets,
- management of repository changesets vs. PIDE edits.

Recall that PIDE is centered around first-class edits with formal markup [7]. Thus it is natural to connect this with Mercurial changesets, to scale the editor model towards the persistent history of the SCM. This should also allow to edit multiple versions in the same editor session.

3.2 Building: batch-mode

Building sessions for the Prover IDE is implicit: users do not have to worry about it. 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. 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`: it 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).

Further scaling could mean to reduce wait-times, by allowing multiple sessions to be built independently in the IDE: it would allow the user to continue working with the active session. It could also help to provide a GUI panel with options for the build process: e.g. to trim sessions to what is required in the IDE right now, not what is statically specified in session ROOT entries.

Command-line build is the main operation for Isabelle + AFP library management: it works via `isabelle build` from the terminal, or via the Isabelle/Scala function `Build.build` (e.g. in the Scala Console of Isabelle/jEdit). There are also some options for batch-mode document preparation, but in the longer run it is better to make this independent of the actual build process (e.g. by using a stored database of PIDE markup to generate LaTeX).

The implicit IDE build process actually uses the same `Build.build` function internally. Here are typical examples for command-line invocation of `isabelle build` and `isabelle jedit` that may shorten wait-times when working with big sessions, by fine-tuning the session context of the IDE:

- explicit build of session images and PDF documents:

  ```
  isabelle build -b HOL-Analysis

  isabelle build -b HOL-Probability
  ```

- implicit build of PIDE session images:

  ```
  isabelle jedit -S Deep_Learning -A HOL-Probability

  isabelle jedit -S Deep_Learning -A HOL
  ```

The `isabelle build` tool provides numerous command-line options, see chapter 2 of the Isabelle System Manual [5]. 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 (e.g. 24 cores), this allows to build the full Archive of Formal Proofs in approx. 1 h elapsed time, when excluding the `slow` group.

Note that advanced applications of `isabelle build` should avoid fancy shell scripting, but use the underlying Isabelle/Scala functions directly: `Build.build`, `Sessions.load_structure`, `Sessions.deps` etc. This already improves performance, because an already running Isabelle/Scala process is faster in processing the build dependencies. Repeated JVM cold-starts and renewed "just-in-time compilation" should be avoided.

Batch-builds are traditionally perceived as a closed process, to produce the required heap images for a PIDE session, or to test sessions from a library. For further scaling, 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.
2. A finished IDE session could be saved as a heap image, for re-use in other IDE sessions or batch-builds.

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

Off-line build of historic versions works via `Admin/build_history` and the corresponding Isabelle/Scala function `Build_History.build_history`. There is also `Build_History.remote_build_history` to invoke `Admin/build_history` remotely over an SSH connection.

The key idea is to use a recent Isabelle version to manage the build process of historic versions from the repository, back to the changeset with the tag `build_history_base` (`20126dd9772c` from 08-Jan-2013). It means that there is no rush to test incoming repository changes immediately: tests may be postponed or repeated later on. This removes some burden from the nightly-build infrastructure for Isabelle + AFP: temporary failure of this service does not lead to gaps in long-term performance measurements; missing versions will be run eventually.

The `build_history` tool also supports correlated multicore builds of the same versions on the same hardware (e.g. for threads = 1, 2, 4, 8). Thus it provides routine performance figures for further scaling of multicore performance.

Actual build jobs are run remotely via SSH, using internal `SSH` module of Isabelle/Scala. This makes it easy to invoke remote processes that return results (build logs) in a controlled manner, without assuming shared file-spaces.

Build log data is passively archived as log files (with formats slightly changing over the years) and actively managed in one big single PostgreSQL database. The cumulative database of Isabelle + AFP since 2002 has an approx. size of 5 GB, but note that detailed ML statistics are stored as blobs with XZ compression.

Database access is presently limited to the Isabelle cronjob that oversees nightly builds, which are very important for long-term performance measurement. A static snapshot of build log data from the past 30 days is published as https://isabelle.sketis.net/devel/build_log.db — it can be inspected e.g. with SQLite Browser or used by other tools using a suitable SQLite library.

Further scaling could mean to publish relevant performance data via an HTTP server that is implemented in Isabelle/Scala: it would provide limited access to some of the data, but not arbitrary database connections to anybody on the Web. A future version of `isabelle build` could query the server about previous timing of sessions, even individual theories. This could be used for advanced scheduling of multiple process that run multiple threads, without asking the user to guess magic command-line parameters.

Build status reports and visualization is the final stage of the nightly Isabelle cronjob. Its results are published on https://isabelle.sketis.net/devel/build_status. The Isabelle/Scala implementation uses SSH port-forwarding for the PostgreSQL

server and generate static HTML pages with some gnuplot charts (see the function `Build_Status.build_status`).

Further scaling could mean more dynamic reports, e.g. by an HTTP server or in the Prover IDE. Easily accessible build status reports might also motivate users to trim down sessions in AFP that are more heavy than actually required (due to redundant imports are too detailed hierarchies of session images).

3.3 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 so far, which resembles the syntax highlighting in Isabelle/jEdit, e.g. see https://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[19]. Visual Studio Code 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 3 on page 17 (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.

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

[19] https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2017

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. Full markup information is managed by the Isabelle/ML prover process and the Isabelle/Scala PIDE process, which are running 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, but the heavy Java IDE front-end is missing.
   (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 could be translated to JavaScript via Scala.js[20]. 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 functionality and robustness.

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 end-users to deliver exactly one well-defined version.[21] 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 tefox or Chromium browsers.

[20] https://www.scala-js.org

[21] The Isabelle/VSCode 1.0 experiment has already suffered from ongoing changes of VSCode and add-on modules. Despite official version schemes for Node.js modules, small changes can have big impact. Long-term experience with Isabelle/jEdit shows that all vital components need to be bundled as one static application, for end-users to download and run without further ado.

4 Conclusion

Scaling Isabelle technology is not new. That demand has been with us from early on, when Isabelle applications grew beyond small academic experiments. I have felt the weight and burden of AFP for the first time in 2006, when it was tiny compared to today. Thanks to continued improvements of the Poly/ML platform by David Matthews, and reorganization of Isabelle system technology around it, we have so far managed to cope with the wealth of user contributions.

It is definitely possible to continue scaling the Isabelle technology in the near future, but that requires action to be taken in various areas as I have sketched in this paper. Some further details are given in the the document "Scaling Isabelle Proof Document Processing" from December 2017[22], notably section 3 "Technical approaches to scaling" — this work was financially supported by *Mu Operator GmbH, Frankfurt am Main.*

References

[1] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[2] D. Matichuk, T. C. Murray, and M. Wenzel. Eisbach: A proof method language for Isabelle. *J. Autom. Reasoning*, 56(3):261–282, 2016.

[3] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[4] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

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

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

[7] M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In G. Klein and R. Gamboa, editors, *5th International Conference on Interactive Theorem Proving, ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014.

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

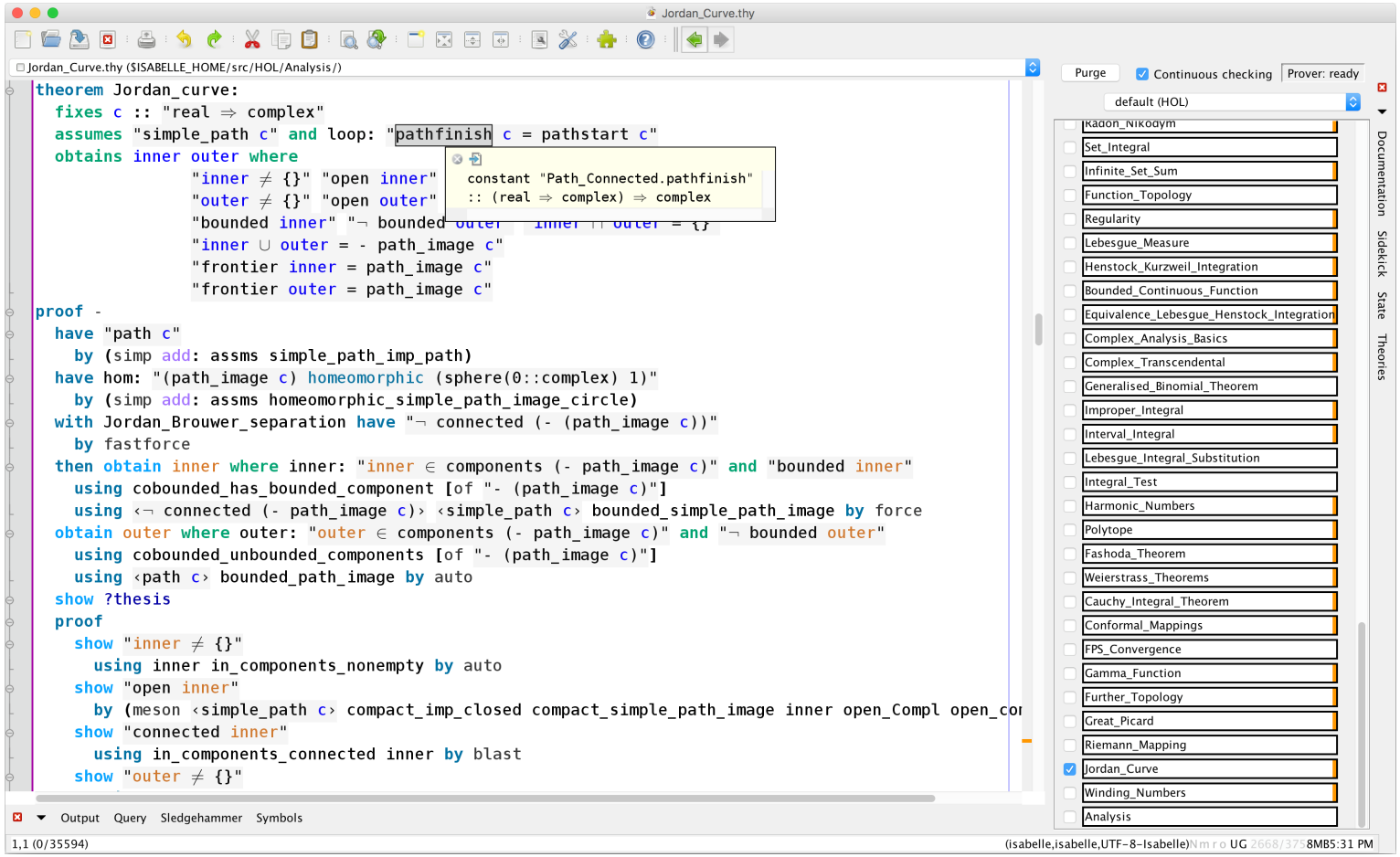

Figure 2: Session `HOL-Analysis` within Isabelle/jEdit (ML process: 2.8 GB, JVM process: 3.5 GB)

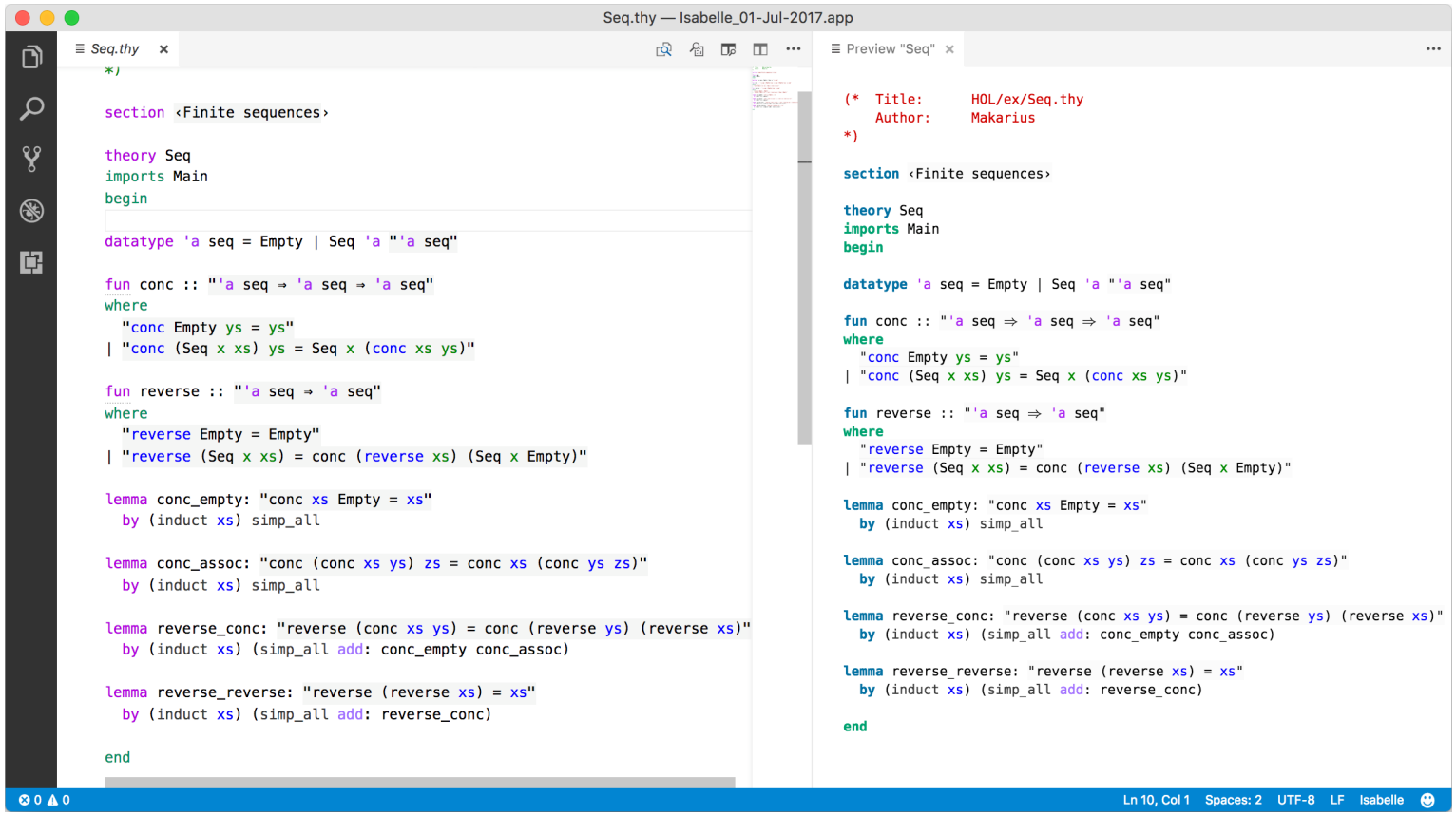

Figure 3: Isabelle/VSCode Prover IDE with built-in HTML preview (Chromium)