Isabelle/VSCode in January 2017

Makarius Wenzel

January 26, 2017

Abstract

This is a report on the results of working with Visual Studio Code over some weeks, in order to connect it to the Isabelle/PIDE infrastructure via the Language Server Protocol. This newly emerging editor by Microsoft (for all platforms and all users) is a nice basis for semantic processing of proof documents, but more work is required to approach the sophistication of the established Isabelle/jEdit front-end.

1 Isabelle/VSCode

Visual Studio Code or **VSCode** is an open-source project[1] by Microsoft, with the slogan "*Code editing. Redefined.*" It is a cross-platform application for Linux, Mac OS X, and Windows — based on the Node.js engine with browser technology for the GUI. The idea is to combine 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*[2] of extensions for the VSCode editor with add-ons for many languages.

Isabelle/VSCode is an extension for VSCode that uses the Isabelle/PIDE programming interface to provide semantic languages services. It consists of a small TypeScript module within VSCode and an external Isabelle/Scala process that runs the *VSCode Language Server Protocol* (a subset of version 2.x[3] with some additions from the emerging version 3.x[4]). The back-end of Isabelle/VSCode is part of the Isabelle distribution (January 2017 or later, e.g. version 09b872c58c32).

As long as there is no official release, the development snapshots from http://isabelle.in.tum.de/devel may serve as approximation. See also http://www4.in.tum.de/~wenzelm/Isabelle_26-Jan-2017 for a specific version corresponding to the VSCode extension *Isabelle 0.6.0*.

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

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

[3] https://github.com/Microsoft/language-server-protocol/blob/master/versions/protocol-2-x.md

[4] https://github.com/Microsoft/language-server-protocol/blob/master/protocol.md

The semantic language services are illustrated in fig. 1 (control-hover over formal entities) and fig. 2 (completion of formal entities). Diagnostic messages (errors, warnings, information messages) are shown as well: all of this is updated continuously as the user is typing.

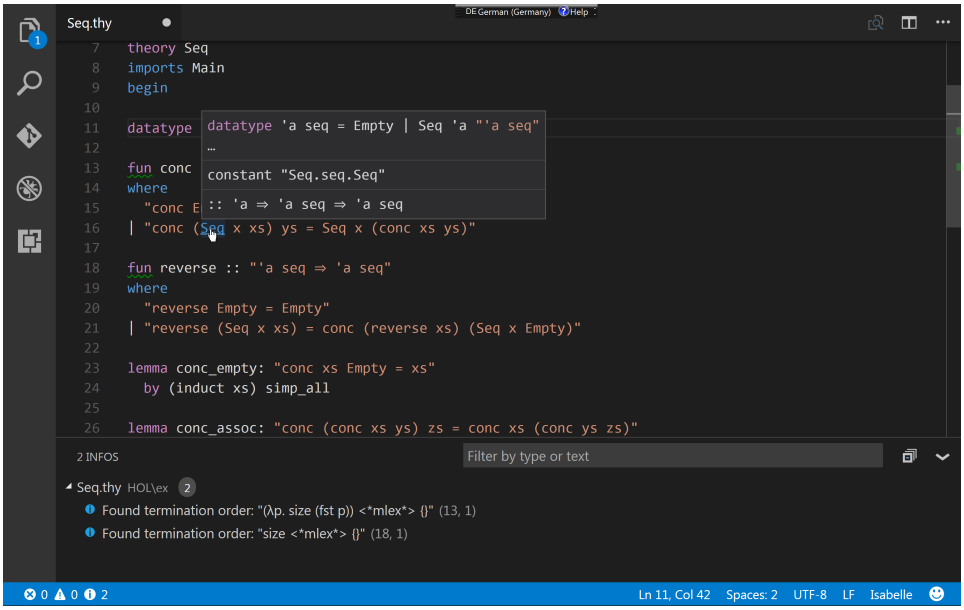

Figure 1: Semantic information by hovering over formal entities

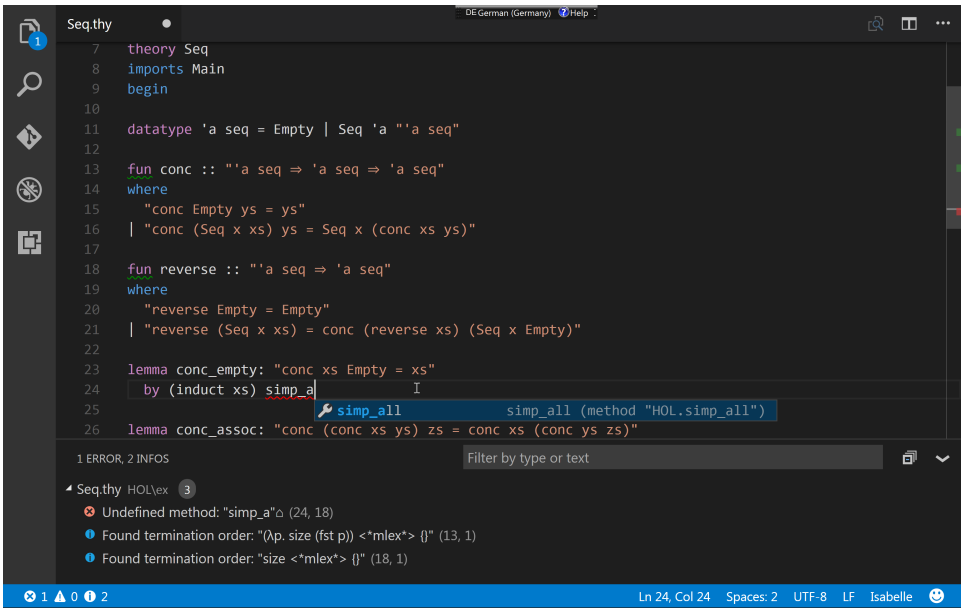

Figure 2: Semantic completion for partial (broken) input

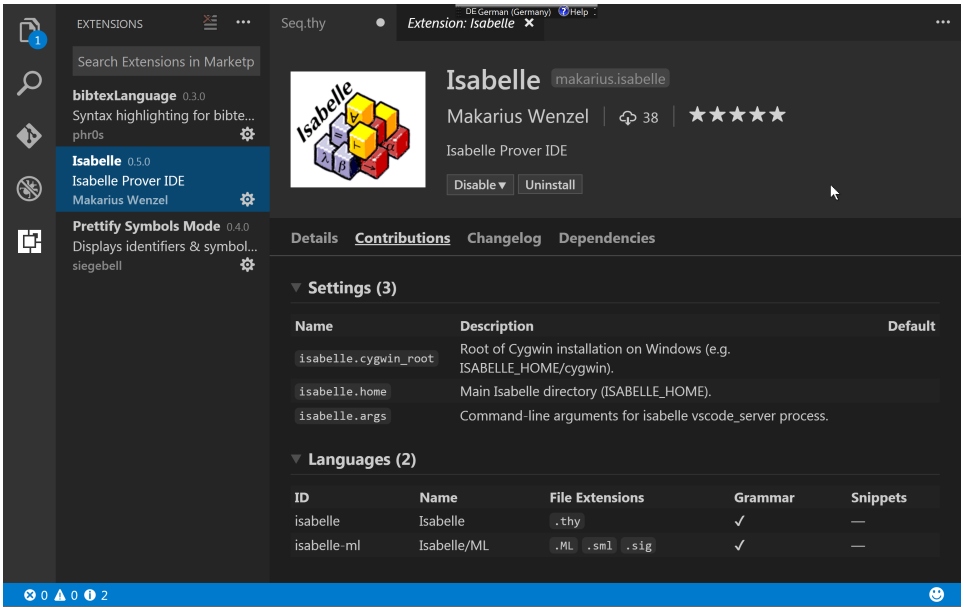

Figure 3: The VSCode editor with the Isabelle extension installed

2 Running Isabelle/VSCode

Here follows a brief explanation of how to download, install and run the Isabelle/VSCode application.

- Download a recent Isabelle development snapshot from http://isabelle.in.tum.de/devel or the particular test version http://www4.in.tum.de/~wenzelm/Isabelle_26-Jan-2017.
- Unpack and run the main Isabelle application as usual, to ensure that the logic image is built properly and Isabelle works as expected.
- Download and install VSCode from https://code.visualstudio.com
- Open the VSCode *Extensions* view and install the following (fig. 3):
  - *Isabelle* (e.g. version 0.6.0 for `Isabelle_26-Jan-2017`)
  - *Prettify Symbols Mode* (optional, but recommended). It is required for nice Unicode rendering of Isabelle symbols, with manual configuration as explained below.
  - *bibtexLanguage* (optional). It is merely required for working with `.bib` files within the Isabelle session, e.g. when following `@{cite` *ref*`}` links in Isabelle document sources.

- Open the dialog *Preferences / User settings* and provide the following entries in the second window, where local user additions are stored: `isabelle.home` on all platforms and `isabelle.cygwin_home` on Windows. For example:

  Linux

  ```
  "isabelle.home":
    "/home/makarius/Isabelle_26-Jan-2017"
  ```

  Mac OS X

  ```
  "isabelle.home":
    "/Users/makarius/Isabelle_26-Jan-2017.app/Isabelle"
  ```

  Windows

  ```
  "isabelle.home":
    "C:\\Users\\makarius\\Isabelle_26-Jan-2017",
  "isabelle.cygwin_home":
    "C:\\Users\\makarius\\Isabelle_26-Jan-2017\\contrib\\cygwin"
  ```

- Restart the VSCode application to ensure that all extensions are properly initialized and user settings are updated. Afterwards VSCode should know about `.thy` files (Isabelle theories) and `.ML` files (Isabelle/ML modules).

The Isabelle extension will be initialized on the first opening of some Isabelle file. It requires a few seconds to start up, with a small popup window saying "*Welcome to Isabelle ...*". If that fails, there is probably something wrong with the above user settings, or the Isabelle distribution does not fit to the version of the VSCode extension from the Marketplace.

After successful startup of the Isabelle/VSCode background process, continuous checking is always on. This may involve some delays depending on the total amount of sources accessed by the editor session. Note that theory dependencies are resolved automatically behind the scenes: it may result in processing much more text than seen in the editor.

A suitable example from the Isabelle distribution is `$ISABELLE_HOME/src/HOL/ex/Seq.thy` — its semantic annotations by Isabelle are shown in fig. 1 and 2. It is also possible to work with Isabelle/ML files, e.g. those of Isabelle/Pure itself by opening `$ISABELLE_HOME/src/Pure/ROOT.ML` and clicking on some of the files that are used via **ML_file** commands. Here the volume of formal annotation already poses a challenge to the present Isabelle/VSCode setup; the editor stops displaying diagnostic messages after some limit is reached. Nonetheless, the ML files are annotated by the Poly/ML compiler, with inferred types for sub-expressions etc.

The extension **Prettify Symbols Mode** requires separate configuration via VSCode User Settings. The Isabelle distribution contains `$ISABELLE_HOME/src/Tools/VSCode/extension/isabelle-symbols.json` — its definition of `prettifySymbolsMode.substitutions` needs to be copied carefully into the local user settings file, such that it coexists with other language configurations for that extension.

A full restart of VSCode and reload of Isabelle sources might be required to see Isabelle symbols like `\<forall>` rendered as ∀. Note that this is just an optical illusion: input sources remain unchanged in their ASCII representation of symbols. That is a potential source of problems, because the Isabelle process outputs symbols directly in this Unicode form: it may cause some confusion when output is copied back into input and files are saved later.

3 Notes on the implementation and limitations

3.1 Isabelle extension for VSCode

From the perspective of VSCode, the Isabelle extension is the main entry point. It consists of:

- the canonical package definition `$ISABELLE_HOME/src/Tools/VSCode/extension/package.json`;
- some `grammar` and `language` files that approximate the syntax of Isabelle theories and ML files statically;
- the TypeScript module `$ISABELLE_HOME/src/Tools/VSCode/extension/src/extension.ts`: this is the client side of the Language Service provided by Isabelle/VSCode.

The VSCode editor provides many more possibilities to augment it for specific needs[5], but the present project was mainly to explore the Language Server Protocol.

Note that static grammar and language definitions are not ideal: Isabelle syntax depends on theory imports: new commands may be defined in user libraries. Isabelle/jEdit has managed to cope with that after several years, such that each buffer has its own syntax that is dynamically provided by Isabelle/PIDE. Analogous tricks would have to be devised for VSCode to make it work beyond statically generated syntax files.

[5] https://code.visualstudio.com/docs/extensionAPI/extension-points

3.2 Isabelle/VSCode language service

The main IDE functionality is provided by the Isabelle/Scala module `isabelle.vscode.Server` (see also `$ISABELLE_HOME/src/Tools/VSCode/src/server.scala`). It is a command-line tool that implements a JSON-RPC protocol over stdin/stdout.

The implementation is relatively straight-forward, both in terms of Isabelle/PIDE and the VSCode Language Server Protocol[6]. There are no special tricks that are inevitable for a fully-featured application — one that scales to big Isabelle applications.

Here follows a brief account of limitations that require further elaboration.

- Treatment of Isabelle symbols in a coherent way: Isabelle/jEdit does that by registering its own character encoding, such that the editor loads and saves files uniformly, while wowing nice Unicode glyphs to the user.
- Convenient input methods for Isabelle symbols, as seen in Isabelle/jEdit.
- More scalable theory processing by informing the back-end about the *editor perspective*, i.e. the set of open files with their visible line ranges. This is important to keep Isabelle/PIDE processing reactive and scalable: formal processing only happens where the user is actually looking. Presently all sources (and resolved dependencies) are processed all the time, which can lead to considerable delays.
- More scalable treatment of "diagnostics" in VSCode terminology (errors, warnings, information messages). The VSCode protocol only allows to replace *all* messages for each file in turn, but this needs to be done more incrementally to accommodate the hundreds or thousands of messages that proof processing typically produces.
- Some means for proof state output in a separate window. It is possible to do some structured Isar proofs just with the annotations of the main editor buffer, but peeking into the internal state is definitely important. Presently, all state output is suppressed; even the explicit **print_state** command remains silent.
- Pretty-printing of message output, potentially with formal links, as seen in Isabelle/jEdit. Presently, the output is plain-text only with a fixed margin – like for typical programming language compilers. VSCode supports Markdown structure routinely, but that is not used in Isabelle/VSCode at the moment. Computed line breaks are not

[6] https://github.com/Microsoft/language-server-protocol

covered by this model anyway: even the standard language setup for TypeScript shows badly formatted messages, depending on different popup window sizes.

- Visual feedback on the processing status of individual prover commands (i.e. long-running and potentially non-terminating tasks). The user needs to know where critical tasks are running — programming language IDEs usually don't have this problem.

- Some prover session overview of the processing status of all theories, similar to the *Theories* panel in Isabelle/jEdit. The user needs to keep an overview of the whole project.

- Etc. etc. — there are open-ended possibilities to make the best out of each editor platform to fit it tightly to Isabelle/PIDE.

As a starting point for further exploration of the VSCode editor — beyond the Language Server Protocol — the following extension might be helpful: https://github.com/siegebell/vscoq. That is an imitation of Proof General and CoqIde within the VSCode editor, based on the XML-version of its REPL protocol.

4 Conclusion

Overall, VSCode looks like a promising editor framework for all platforms and all users. The Language Server Protocol has big potential to reduce the quadratic connectivity problem of IDE front-ends vs. language back-ends.

Some limitations of the protocol might either disappear spontaneously, as has already happened in the transition from version $2.x$ to $3.x$, or could be overcome by protocol extensions. The VSCode reference implementation for the Language Client makes it easy to install handlers for custom protocol messages. So in some sense, protocol extensions are already part of the game.

After further iterations and inevitable special tricks to make things work smoothly and scalable, one could also approach the protocol maintainers, and tell them about the specific needs for Prover IDE support.

What we have here is essentially a *Semantic IDE*, where language processing also involves evaluation with potential non-termination. This requires some means of control for the user and more policies for the connection of front-end vs. back-end. Particularly important is a clue about the *editor perspective*, i.e. the visible part of the workspace and the cursor position.