Getting Started with Isabelle/jEdit

Christian Sternagel*

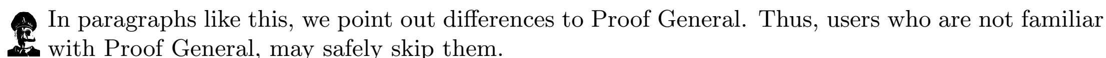
JAIST, Japan
c-sterna@jaist.ac.jp

Abstract

We give a beginner-oriented introduction to Isabelle/jEdit, providing motivation for using it as well as pointing at some differences to the traditional Proof General interface and current limitations.

1 Introduction

Before starting, let us clarify our goals. This document is intended as a short introduction to Isabelle/jEdit (as shipped with `Isabelle2012`) from a new user's perspective. We highlight some differences to Proof General and common pitfalls for new users (mostly issues that have been collected from the Isabelle mailing lists). We do *not* give any technical or implementation details (for such, refer to [1, 3, 5, 4, 2] and the Isabelle documentation). Moreover, our intended audience are newcomers to Isabelle/jEdit (that is, we have nothing to say that people could not easily find out on their own, but hope to save some people some time).



In paragraphs like this, we point out differences to Proof General. Thus, users who are not familiar with Proof General, may safely skip them.

Points that may need special attention are marked like this.

We use the following conventions: When indicating a menu like A→B→C we mean: "Click on submenu C in submenu B of menu A." When indicating keyboard shortcuts like C-s we mean: "Keep key C pressed while pressing key s." Here, C refers to the *control* key (usually labeled with Ctrl on PC keyboards; on Mac OS this is actually COMMAND).

The remainder is structured as follows: First, in Section 2, we argue why we think that Isabelle/jEdit is more suitable as a user interface to Isabelle than the traditional Proof General, but also give some current limitations. Then, in Section 3, we explain how to obtain and start Isabelle/jEdit and give an example walkthrough. The next section (Section 4) is concerned with some typical use cases. Finally, we conclude in Section 5.

2 Why Isabelle/jEdit is Awesome

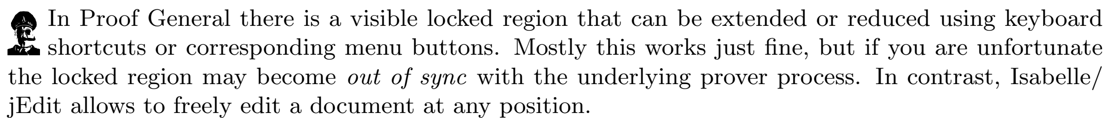
The most important point (in our opinion) is that Isabelle/jEdit (or rather the technology[1] that makes it possible to use the editor jEdit as an interface to the proof assistant Isabelle) provides an interaction model that is broadly known and used nowadays (e.g., in word processors and integrated development environments for programming languages). Lets call it the *continuous model*. The user sees and *freely* edits a document and all the heavy machinery of Isabelle

*Supported by the Austrian Science Fund (FWF): J3202.

[1] Isabelle/jEdit is a client to the PIDE framework [5]; there may be other clients (e.g., web-based or as part of an existing IDE, like Eclipse).

asynchronously runs in the background and supplies *source text* with semantic information that is again presented in ways people are used to nowadays (highlighting, hyperlinks, tooltips, wavy underlines, etc.).

The traditional interaction model, in contrast, is more like a command-line interface, we sequentially enter commands (one at a time) and wait for the corresponding output. Since this would be exceedingly tedious whenever we wanted to change something we entered very early (essentially, we have to enter everything again), this kind of interaction is usually combined with source text. Such that instead of typing everything again, we just tell the system which part of the source text has to be reloaded. However, in order for this to work, we need some kind of *locked region* (the part of the source text that has already been processed) which we are not allowed to edit. Or alternatively, reload everything after each change.



In Proof General there is a visible locked region that can be extended or reduced using keyboard shortcuts or corresponding menu buttons. Mostly this works just fine, but if you are unfortunate the locked region may become *out of sync* with the underlying prover process. In contrast, Isabelle/jEdit allows to freely edit a document at any position.

At this point you may wonder: "What's the big deal about the continuous model?" Since it may seem that the only difference to the traditional model (lets call it the *sequential model*) is that we do not have to manage a locked region manually. Well, the main point is that the sequential model is inherently sequential, whereas the continuous model offers plenty of space for parallelization. Thus, Isabelle/jEdit does not only provide a more modern (and for many potential users more familiar) user interface but additionally facilitates (again, the real reason is rather the underlying technology) to seamlessly make use of multi-core architectures.

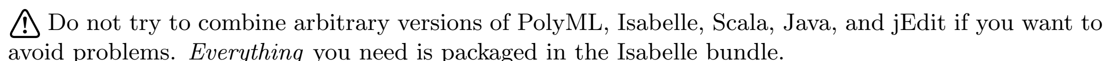
Having said this (and that is the reason why Isabelle/jEdit is indeed awesome), bear in mind that we are talking about brand new technology here (which was moreover mostly developed by a single person; cf. the references). Hence, it comes as no surprise that there are still some small quirks and deficiencies. Most notably, there is no real tutorial for Isabelle/jEdit and most menus that where available in the traditional interface are still missing. We will have more to say about this in Section 4. Nevertheless, Isabelle/jEdit *is* ready for production use *now*.

3 Get Rolling

Before we can use Isabelle/jEdit, we have to download it. Since it is contained in the official Isabelle release, we download that from:

> http://isabelle.in.tum.de or http://www.cl.cam.ac.uk/research/hvg/Isabelle/
> or http://mirror.cse.unsw.edu.au/pub/isabelle/

The correct version for your operating system and architecture should be provided by the big green button and can be installed to an arbitrary directory `ISABELLE_HOME`.



Do not try to combine arbitrary versions of PolyML, Isabelle, Scala, Java, and jEdit if you want to avoid problems. *Everything* you need is packaged in the Isabelle bundle.

To start Isabelle/jEdit from a command line (e.g., under Linux) use (something similar to)

```
ISABELLE_HOME/bin/isabelle jedit
```

(Under Mac OS click on the *Isabelle2012.app* icon and then select *Isabelle/jEdit* instead of the default *Proof General*. Under Windows click on *Isabelle2012.exe*.)

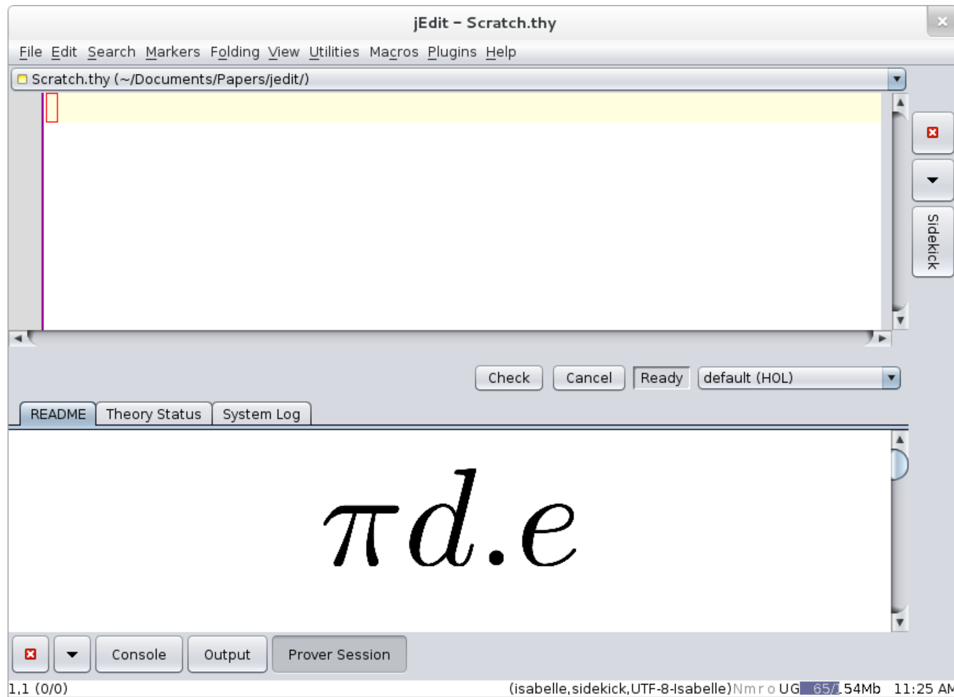


Figure 1: Isabelle/jEdit when first started.

In Figure 1 you see how Isabelle/jEdit presents itself after being started the first time (we just reduced the initial window size in favor of readability). On the top half we have the *main buffer*, this is where our source text is shown and editing takes place. At the bottom of the lower half we have (among others) buttons *Output* and *Prover Session* which determine the content of the panel directly above of them. By default *Prover Session* is selected. When selecting *Output* the panel shows the *output buffer*, this is where messages from the *current command* are to be found, where the current command is determined by the cursor position in the main buffer (i.e., by default the output buffer shows the message(s) corresponding to the command on which the cursor is positioned).

⚠ Under *Prover Session* we have a panel containing the tabs *README* and *Theory Status*. The former is of general interest (since it lists some limitations and workarounds as well as known problems) and the latter can always be consulted to check whether Isabelle/jEdit agrees with you on what files are loaded (and how much of them), where problems are indicated in red.

⚠ Prover Session also contains a drop-down list where it seems to be possible to select a certain logic image (where the default is *default (HOL)*). This does, however, not work at the moment. Instead you have to restart Isabelle/jEdit like

```
isabelle jedit -l Image
```

where `Image` is the name of the desired logic image and you have to make sure that the selection is *default (HOL)* (otherwise, the `-l` flag will be ignored).

To start using Isabelle/jEdit let us formalize a simple fact about lists. Before we can do so, we need to start a *theory* (which is Isabelle parlance for a source file that bundles definitions and facts under a common name). We start by typing "`theory Test imports Main begin`" into the main buffer, which tells the system to start a theory with name *Test* on top of the theory named *Main*, the default starting point for Isabelle theories (i.e., providing some kind of "standard library" of definitions and facts). At this point you may notice an error message in the output buffer (which is additionally stressed by showing a prohibition sign in the left margin as well as wavy underlining the word "`theory`" in the input.

```
Bad file name "Scratch.thy" for theory "Test"
```

This just happened because the default file name for theories is `Scratch.thy` and can easily be resolved by saving (either File→Save or C-s) the current file under the name `Test.thy`.

⚠ A theory with name *Name* has to reside in a file with name `Name.thy`. The Isabelle convention is to capitalize theory names and separate words by underscores, e.g., `Complete_Partial_Order` rather than `completePartialOrder`, `complete_partial_order`, etc.

Now, let us come back to the announced fact about lists: when we combine the head and the tail of a non-empty list, we obtain the same list again. To make it a bit more interesting (and informative), we define our own functions for computing the head and the tail of a list as follows:

```
fun head :: "'a list => 'a" where "head (x # xs) = x"
fun tail :: "'a list => 'a list" where "tail (x # xs) = xs"
```

There are already a few things to note here. When defining `head`, a warning occurs (which is additionally stressed by showing a light bulb sign in the left margin as well as wavy underlining the word "fun" in the input)

```
Missing patterns in function definition:
head [] = undefined
```

which tells us that our function is not defined on input `[]`. This is not an error but just a warning, since we could intend our function to be undefined in this case (as we indeed do). A similar warning is issued for `tail`. Further note that the status of an identifier is indicated by its color: free variables are rendered blue, bound variables green, defined constants black, etc. Moreover, many entities allow to jump to their definition by holding C pressed while left-clicking on them with the mouse. For `head` and `tail` this means that we end up in the line containing the corresponding `fun` keyword (even if that line would be part of a different file). We can test this behavior by C-left-clicking `#` in the definition of `head` which brings us to the theory *List* of Isabelle's standard library. More concretely, to the line where the list datatype is defined (since `#` refers to one of the constructors of the list datatype). The fastest way of going back is C-' (control together with backtick, not to be confused with a single quote) or View→Go to Recent Buffer.

Now we prove our little lemma (note that at this point `head` and `tail` are printed black, since they are defined constants now):

```
lemma "~(xs = []) ==> head xs # tail xs = xs"
  by (cases xs) simp_all
```

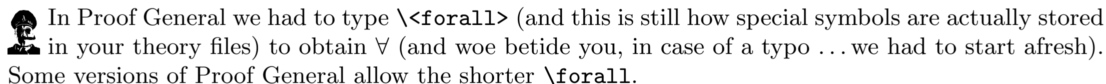
We finish our small theory by adding `end` at the end of the file.

4 How To . . .

In this section we give several typical use cases of things that you might want (or have) to do inside Isabelle/jEdit. It is organized by typical user questions/statements.

4.1 How can I use fancy mathematical symbols?

This is really easy and convenient thanks to the code completion facility of the SideKick plugin. By default, code completion is active (you can change the corresponding settings at Plugins→Plugin Options. . . →SideKick) and works as follows: you just enter what you mean in plain ASCII and if available the system offers you a popup with alternative choices. If you want to close the popup, just keep on typing (if you are at the end of a word just enter a space) or press ESC. If you want to choose an alternative, navigate through the popup using the arrow keys (or the mouse) and press ENTER when you are satisfied. E.g., after typing `==>` (two equal signs followed by a greater than) a popup will show up containing $\Longrightarrow$. Many symbols also have names that are close to the usual LaTeX names, e.g., type `forall` to obtain ∀, `exists` to obtain ∃, etc. Often, you do not even have to enter the full name, a prefix is enough, e.g., after typing `fo` you will obtain a choice between the keyword `for`, the symbol ∀, and the symbol **4**.



In Proof General we had to type `\<forall>` (and this is still how special symbols are actually stored in your theory files) to obtain ∀ (and woe betide you, in case of a typo . . . we had to start afresh). Some versions of Proof General allow the shorter `\forall`.

4.2 There is nothing wrong in my file, but jEdit complains!

It can happen that the parser (which avoids to reparse the whole file for obvious efficiency reasons) of Isabelle/jEdit gets confused by your editing, resulting in a wrong error message. In such a case, the usual workaround is to select a chunk of text surrounding your edit and then cut and paste it (thereby forcing Isabelle/jEdit to reparse the whole chunk). Usually it is a good idea to cut and paste a "complete" chunk of text, e.g., a comment including `(*` and `*)`, a whole lemma statement (not necessarily the corresponding proof), etc.

4.3 Where have all the menus gone?

In Proof General many settings of Isabelle and diagnostic commands could be configured and started via menus. This is currently not supported in Isabelle/jEdit. The workaround is to explicitly change a setting or start a diagnostic command from your main buffer. This is not as tedious as it may sound due to code completion (e.g., a popup containing `find_theorems` already occurs after typing `fi`). The bigger drawback is that we have to memorize all the available settings and commands (since otherwise you do not know what to type). To this end, the Isar reference manual (`isabelle doc isar-ref`) is very helpful, which contains descriptions (and names) of all available settings and commands. For configuration options like `show_types` you additionally have to know how to set them from inside the main buffer, which works by typing something like:

```
declare [[show_types, show_sorts=false, ...]]
```

Arguably the most frequent diagnostic commands are (in arbitrary order):

- `find_theorems` for finding theorems

- `print_cases` for printing the available cases inside an induction or case analysis
- `sledgehammer` to invoke sledgehammer on the current goal (a nice feature is that after sledgehammer reports a metis command in the output buffer, you can just click on it to substitute it for the `sledgehammer` in your main buffer)
- `nitpick` and `quickcheck` for finding a counter example to the current goal

5 Conclusions

In total, we just summarized some of the points that have already been discussed on the Isabelle mailing lists (or elsewhere). Nevertheless, we hope that this short and beginner-oriented introduction may help potential new users out there to get up and running faster than they would have without this document.

A final point for considering to use Isabelle/jEdit instead of Proof General is the following (it is rather practical than ideological):

The Isabelle developers do no longer maintain Proof General for Isabelle and until now nobody else stepped forward to do so.

Acknowledgments. We want to thank the Isabelle community (especially the developers) for always being very helpful (and friendly) on the Isabelle mailing lists and Makarius Wenzel for developing such an awesome tool as Isabelle/jEdit (keep going!).

References

[1] Makarius Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In *User Interfaces for Theorem Provers, UITP 2010*, Electronic Notes in Theoretical Computer Science. to appear.

[2] Makarius Wenzel. READ-EVAL-PRINT in parallel and asynchronous proof-checking. In *User Interfaces for Theorem Provers, UITP 2012*. www.informatik.uni-bremen.de/uitp12/papers/paper-02.pdf.

[3] Makarius Wenzel. Isabelle as document-oriented proof assistant. In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Conference on Intelligent Computer Mathematics, CICM 2011*, volume 6824, pages 244–259. Springer, 2011. doi:10.1007/978-3-642-22673-1_17.

[4] Makarius Wenzel. Isabelle/jEdit – a prover IDE within the PIDE framework. In Johan Jeuring et al., editors, *Conference on Intelligent Computer Mathematics, CICM 2012*, volume 7362. Springer, 2012. To appear. http://arxiv.org/abs/1207.3441.

[5] Makarius Wenzel and Burkhart Wolff. Isabelle/PIDE as platform for educational tools. In Pedro Quaresma and Ralph-Johan Back, editors, *CTP Components for Educational Software, THedu 2011*, volume 79 of *Electronic Proceedings in Theoretical Computer Science*, pages 143–153, 2011. doi:10.4204/EPTCS.79.9.